AD-A050 294 NEW YORK INST OF TECH OLD WESTBURY F/G 9/2
FUNCTIONAL SOFTWARE DEVELOPMENT.(U)
NOV 77 M M DROSSMAN AFOSR-77-3205
UNCLASSIFIED AFOSR-TR-78-0110 NL

| OF |
AD
A050294

END
DATE
FILMED
3-78
DDC

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR-78-0110	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): FUNCTIONAL SOFTWARE DEVELOPMENT.	5. TYPE OF REPORT & PERIOD COVERED: Final rept.
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Melvyn M. Drossman	8. CONTRACT OR GRANT NUMBER(s): AFOSR-77-3205 new
9. PERFORMING ORGANIZATION NAME AND ADDRESS: New York Institute of Technology, Wheatley Rd, Old Westbury, NY 11568	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS: Air Force Office of Scientific Research/NM, Bolling aFB, DC 20332	12. REPORT DATE: 14 Nov 77; 13. NUMBER OF PAGES: 66 (70 p.)
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

S/C 392979

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Functional software development is a system for the development of high-quality digital computer software. It has two components; functional design for design, and functional programming for program implementation. Functional design has been documented and the feasibility of functional programming is demonstrated.

Functional design is a top-down graphical method based on the concepts

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract

of nested-virtual machines and matching program structure to data structure. Programs developed using these techniques consist of a set of functionally cohesive modules whose linkages are automatically handled by the language processor component of the functional programming system. As a result, they should be easily maintained and reliable.

AFOSR-TR- 78-0110

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH

FINAL SCIENTIFIC REPORT

FROM: Melvyn M. Drossman
New York Institute of Technology
Wheatley Road
Old Westbury, NY 11568

TO: AFOSR/PM
Bldg. 410
Bolling AFB DC 20332

Report Date: 14 November 1977

Grant No: AFOSR-77 3205

Signed: Melvyn M. Drossman

Title: Chairman, Computer Science and
Electrical Engineering Technology

FUNCTIONAL SOFTWARE DEVELOPMENT

Melvyn M. Drossman

1. Introduction

Advances in digital computer hardware have made it possible for computers to process programs whose complexity exceeds that which most designers are presently capable of effectively handling. This disparity between the state-of-the-art in the area of computer hardware and that in computer software production has resulted in the generation of a large quantity of inferior computer software.

Recognition of this problem is evidenced by research studies in the area of software quality factors. The purpose of these efforts is to define and quantify measures for the evaluation and comparison of the quality of software products. The entire field of software engineering is a response to this problem and an attempt to advance the state of the art of software development. A number of methods for the creation of better quality software have been reported. A variety of approaches are taken in these methods but most deal with the programming (implementation) phase or the design phase. Considerable effort is also being expended in the areas of testing, verifiction, and validation.

Functional Software Development (FSD) is a system for the design and implementation of computer software which insures that the software product will have certain properties which are associated with high quality. Furthermore, it provides a well-defined approach to the software design process starting at a much earlier phase than most other methods do. This characteristic is important because of the critical nature of the earliest design decisions; if these are not well thought out the ultimte software product cannot be of high quality no matter how excellent the subsequent design and implementation techniques are.

2. Background

It is necessary to have some familiarity with the work that has been done in the area of software quality factors and in software design and implementation methods in order to understand the reasons for, and advantages resulting from, the use of the various procedures incorporated in FSD.

2.1 Software Quality Factors

Efforts in the establishment of software quality factors have been directed toward the definition and quantification of those characteristics of software products which users consider in arriving at a measure of a product's "quality.".

A rather extensive early report by a group at TRW (4) presents a hierarchical set of quality factors in which each factor on a given level is determined by a subset of the factors on the preceding level.

A group of investigators at General Electric present a set of eleven software quality factors (12,13) as part of the preliminary results of a project they are currently engaged in. The values of the software quality factors are determined by the values of a set of software quality criteria. These are found by static measurement, i.e., examination of program listings and other documentation as opposed to dynamic measurement which involves running the program.

Drossman (7) presents ten software quality factors based on those presented in the TRW and General Electric studies and interviews with a number of software users. The results of these interviews indicate that the relative importance associated with each factor tends to be related to the application environment in which the software is to be used. A description of these ten factors follows:

1. Correctness

   Correctness refers to the degree to which a program satisfies the specifications and computes results according to these specifications. This implies satisfactory accuracy and precision. Errors in program performance resulting from errors in specification are not considered in determining the correctness of the software.

2. Efficiency

   Efficiency refers to the efficiency of utilization of computer hardware, including the central processing unit, memory, peripheral devices, and system software. This factor is measured by the quantity of these resources which are used and the amount of time during which they are used. It includes both compilation and other program processing, and the program execution. Efficiency does not refer to the efficiency of designing, writing, testing, or debugging the program; these items are incorporated in other factors.

3. Flexibility

   Flexibility refers to the degree to which a software product can be applied to a variety of applications.

Measurement of this factor includes consideration of the simplicity of modifying the software for the applications as well as the breadth of applications. Software packages consisting of standard modules which can be combined in various arrangements for different applications are more flexible than software designed as a single unit.

Flexibility results in greater efficiency for the designer and programmer because it is more frequently possible to use existing modules; this aspect is sometimes referred to as code reusability. The use of existing modules also enhances reliability because these modules have generally been used to a larger degree and are therefore more likely to be free of errors.

4. Integrity

Integrity is the degree to which the software, including the program and its data bases, are protected from unauthorized access and modification.

5. Interoperability

Interoperability refers to the degree to which a software product may be interfaced to other software for the purposes of building a more comprehensive system, to support the operation of the software product itself, or to support the operation of the software to which it is being interfaced.

6. Maintainability

Maintainability refers to the ease with which a software product may be modified either to correct errors or to satisfy changed specifications. This includes location of the error or segments to be changed, as well as implementation of the change. This factor is extremely important because of the large amounts of money being expended on software; it is estimated that sixty percent of all computer related expenditures by the Department of Defense are for software. (3)

7. Portability

Portability refers to the degree to which a software product designed for a given hardware/software configuration, i.e., machine and operating system, can be moved to another system. Included in this factor is a measure of the ease with which any required modifications can be made.

8. Reliability

Reliability refers to the degree to which a program continues to operate correctly after a period of usage and maintenance and despite changes in its environment, e.g., changes of peripherals, operating system modifications. This last aspect, the ability to operate satisfactorily despite changes and degradation in the environment, is sometimes called robustness.

9. Testability

Testability refers to the degree of confidence in a software product that one can gain by a given amount of effort spent in testing. If it is possible, with a given effort, to exercise all flowchart paths and test for all critical data values, the program has a higher degree of testability than one for which only fifty percent of the paths can be exercised with the same effort devoted to testing.

10. Usability

Usability refers to the ease with which a software product can be used. Considerations included in the measurement of usability are installation effort, effort needed to learn how to use it, effort required for preparation and entry of input, and effort required for interpretation of output.

There is general agreement in the literature that certain approaches to software design and implementation result in improved quality. Many of the methods for software design and production incorporate these approaches. A few of the important general concepts of software design are discussed before considering more specific methods.

Program Modularization

One of the problems in designing complex software is the sheer size of the program and the task which it is intended to perform. The number of functions and their interaction makes

it difficult for designers and programmers to maintain a grasp on the entire program. Separation of the program into modules relieves this problem to a considerable extent by permitting designers and programmers to consider a number of more limited problems, i.e., to separate the problem into a number of smaller problems. Modularization is part of the solution to the problem of designing complex software but, in itself, is not the complete solution. Two problems are inherent in modularization: determining what functions should be incorporated in each module, and designing the interfaces between modules. Many design methods are directed toward the solution of one or both of these problems.

Top-down approach

Software design/programming methods may be characterized as being top-down or bottom-up. In practice, virtually all methods use a combination of both approaches but frequently one will predominate over the other.

In the top-down approach the designer starts by considering the overall problem and separating it into a number of sub-problems each of which will be handled by a subprogram. Then each sub-problem is divided into still smaller sub-problems which are solved by lower level subprograms. The programming effort is handled in a parallel fashion: the main program is written first and is primarily a sequence of subroutine calls; each of the top-level subroutines is written next and these invoke lower-level subroutines which are written next, and so on.

In the bottom-up approach the designer identifies the most basic functions which have to be performed; these are programmed first. Then these modules are integrated to form larger modules and so on until the main program which solves the entire problem has been generated.

It is widely held that methods which are predominantly top-down are better than those which are mainly bottom-up. (2) The major advantages of top-down design over bottom-up design relate to interfacing and testing. Interfaces are attended to in the initial design and form an integral part of the design rather than being elements which are added on after the modules are complete. Testing is simplified because driver routines need not be written to test individual modules. The top-down approach is generally associated with Harlan Mills who used it in conjunction with the chief-programmer team management technique and structured programming to achieve very high productivity on a project to develop an information retrieval system for the New York Times. (1)

Nested-virtual Machines

This is a concept proposed by Dijkstra (2) which results in a top-down modularization of a program. The designer starts by assuming the existence of a machine that contains in its language an instruction which performs the entire function of the proram to be written. The program then consists of this single instruction. The next task is to simulate this virtual machine which really does not exist. The designer divides the function of this virtual machine instruction into a number of major sub-functions and assumes the availability of a machine whose instruction set contains instructions that implement each of these major sub-functions. Using these instructions a program is written which simulates the original virtual machine. This process is repeated, each time assuming the existence of a virtual machine whose instructions are of a lower level than those of the preceding virtual machine which is being simulated, until a point is reached at which the instructions are those of an existing machine or language.

The benefits of this approach, beyond the top-down modularization are that the interfaces between modules are well defined and changes in a given module can be made rather easily because the modules are independent of each other. This approach is classified as a concept rather than a method because the complexity of the data structures preclude direct implementation of the procedures as outlined above. A number of design and implementation methods based on these and other approaches have been developed. A discussion of some of these, especially those relavent to FSD follows:

Structured Programming

In the minds of many programmers structured programming is synonymous with programming without the use of the GOTO statement. This is a consequence of a letter by Dijkstra (6) in which he indicates that programs can be written without the GOTO statement and they will be better structured programs. Structured programming encompasses much more than this however. The important concept of structured programming is to limit the programmer's building blocks so as to achieve simpler program structures. This is intended to insure that designers and programmers can grasp the function and operation of the programs they are working on. Structured programming limits program structures to three constructs (19): sequences of operations, the IF-THEN-ELSE decision and the DO-WHILE iterative loop. Flow charts for these constructs are shown in Fig. 1. In addition to limiting the variety of constructs, the structure constraints also insure that each program segment has a single entry-point and a single exit point.

Structured Design (Composite Design)

Structured Design (also referred to as Composite Design) is a procedure based on the modularization of programs. There are three major concepts involved in this method: (1) module cohesion, (2) module coupling, and (3) program decomposition.

Module cohesion is concerned with the functions encompassed by a single module. Several levels of cohesion are defined. On the lowest level, the various functions grouped into a module are unrelated to each other and are not in any way associateed with each other. On the highest level, called functional binding, the functions included in a module are functionally related to each other. The objective of Structured Design is to try to achieve as high a level of cohesion, or binding, in each module as possible.

Module coupling is concerned with the interfaces between modules. Several levels of module coupling are defined; the greater the inter-connections and interactions between modules, the higher their coupling. The objective in Structured Design is to achieve as low a level of coupling between modules as possible.

The overall aim in Structured Design is to have each program function localized in a single module with as little interaction between functions as possible. This permits the designer to concentrate on a single module at a time and the programmer to write each module as a relatively independent functional entity. This functional isolation of modules facilitates both testing and maintenance.

Program decomposition is the process of partitioning a problem into subproblems for which program modules are written. Graphical representations, called structure charts, are used to show the hierarchical modules structure and the inputs to each module. Two prototype program structures are presented: one is an input-process-output paradigm and the other is an assembly line type of structure.

Jackson Method

The central concept of the Jackson Method (10) is that program structure should match data structure. It is claimed that such an approach results in effective program structures which do not depend on the cleverness of the program designer. It is reasonable that program structure and data structure should be similar. The designer is not dependent on inspired insight into the program modularization since the data structure, assuming it is known, provides the basis for the program structure.

Other Methods

Various additional software development methods which are related to FSD, but to a lesser degree than the preceding ones, are described briefly in the remainder of this section.

Higher Order Software (9) is concerned with the problem of insuring that intermodule linkages are correct. This problem is one of the key problems in writing modularized programs.

Modularization based on levels of data abstraction (11) is another approach to the problem of program design. Its influence is seen in the input-process-output decomposition used in Structured Design. The major concept is to use modules which perform input and output and are separate from the processing module. The input and output modules perform preprocessing and postprocessing, respectively, so that the module which performs the principal processing routine is not affected by details of data formats.

HIPO (16) is a method for graphic documentation of the top-down design of a software system. It uses a set of charts, augmented by notes, which show the various levels of program decomposition based on an input-process-output model. A graphical index is used to relate the charts to each other.

The use of program design languages and requirement languages (5,8) is another approach to the problem of software design. Functional specifications are written for each module. Subsequent phases of the design, development and programming process are guided by these specifications and the results of each phase are checked against these specifications for completeness and consistency. This approach is intended to facilitate computer-aided verification and validation.

3. Functional Software Development Components

Functional Software Development is a system for the design and implementation of high-quality software. The major components of the system are shown in Fig. 2.

3.1 Functional Design

Functional Design (FD) is a method for designing a program or system of programs using a top-down approach. The method is based on two key concepts: the nested-virtual machine concept and the matching of program structure to data structure. Graphical representations of the various program levels, called Functional Development Charts (FDC's), are used

in the problem decomposition. A set of Functional Design Procedure guide this decomposition. Each FDC consists of three elements: the Development Graph, Notes, and the Sequence Graph.

Functional Design can be used by itself to design high-quality programs but the total software development process is greatly enhanced by the use of Functional Programming.

3.2 Functional Programming

Functional Programming (FP) is a method for implementing a program designed using FD. It greatly simplifies the programming process and improves the correctness of the resulting program. The two methods, FD and FP, comprise a synergistic combination.

Functional Programming consists of a language, called Functional Programming Language (FPL), and a processor for it, called the Functional Language Processor (FLP). The language consists of three major elements: a basis language which is an existing language or variant of one, the Data Description (DAD) which is a unified and comprehensive description of the program data, and functional functions and notation which provides for program modularization.

Functional Design is a well developed method which has been applied to the design of a number of systems. Functional Programming is not nearly as well developed; the basic requirements have been defined and some preliminarry consideration has been given to the Functional Language Processor in terms of its feasibility.

Functional Design is a method for the design of high-quality software. A body of procedures are used to create a set of Functional Development charts that are related to the ultimate program in much the same way that a schematic diagram is related to a circuit.

4.1 Characteristics and Concepts of Functional Design

The two key concepts which provide the basis for Functional Design are the nested-virtual machine concept and the approach wherein program structure is related to data structure. Functional Design provides a design implementation of the nested virtual machines as a family of functional sub-programs. The modularization is determined by the data structures so that the program structure is not only related to, but is actually determined by, the data structure.

Furthermore, the function of the modules are dictated by the information flow between input and output. The term "data-directed program development" best summarizes the direct influence of the data structure on the program structure. Data-directed program development, in addition to providing a firm basis for program modularization, also guarantees functional cohesion of the modules as defined in Structured Design (17). This is the highest level of module cohesion and helps to assure a high level of software quality.

Functional Design and Functional Development Charts in particular, provide the designer with a tool which may be compared with a zoom camera lens. The designer starts the top-down design procedure by looking at the overall program requirements and identifying the major data elements and the information flow between them. The "lens" is then zoomed in to take a more detailed look at some aspect of the problem by unfolding, or defining, the data structure and resulting flow of information in more detail. As a graphical representation of the top-down decomposition of a system, Software Development Charts are similar to the charts used in the Structured Analysis and Design Technique by SofTech Inc. (15) although the charts are quite different in other respects. There are also similarities between Functional Development Charts and the charts used in the HIPO method.(16) Some of the elements of the HIPO method did motivate the use of corresponding elements in FSD but the FDC's are quite different from HIPO charts.

The concept of level of data abstraction (11) has had impact on Structured Design and various other methods. This concept is not incorporated in FD but the two are compatible; FD does not guarantee, nor does it preclude, levels of data abstraction in programs designed using it.

4.2 Use of Functional Design

FD is intended for use in the very earliest stages of software development. The use of requirements languages (8) to specify what a program is to do sometimes appears to encroach upon the designer's options by specifying how something is to be done. This is a poor practice because the earliest design decisions provide the foundation for the entire design and hence are very critical; if they are made without adequate consideration they may well prevent the design of a software product having a quality level which could have been realized.

FD provides the designer with the tools to specify a data structure and associated program to implement given functional

requirments. The influence of data structure on program structure makes it desirable to leave the specification of the data structure to the designer insofar as this is feasible. Even, when physical data formats are given, the designer can frequently overlay various conceptual structures on them resulting in programs which are quite different. Experience has shown that program efficiency is very sensitive to the conceptual data structuring. The definition of the conceptual data structure seems to be the most critical phase of the design process and the one which is most dependent on the designer's judgement. This is the activity which should be given the major part of the designer's creative effort; the rest is almost automatic using FD. Perhaps the most important characteristic of FD is that it centralizes the important matters of judgement in the design process into the single problem of deciding upon a conceptual data structure.

4.3 Overview of Functional Design

The design process, using FD, proceeds through a number of levels, each level corresponding to one of the nested-virtual machines. The top level identifies the major data components and their interaction as a single function corresponding to the single instruction executed by this virtual machine. Each subsequent level defines the functions of the preceding level using a number of sub-functions. Each function of the preceding level, except those with a terminal definition, is specified by a Functional Development Chart.

The Functional Development Chart consists of three main parts. The Development Graph, which is the first part and is always present, shows the information flow between the data elements involved in the function. Nodes represent data elements and directed arcs represent information flow or processing. The Notes, which is the second part, and is optional, is a set of notes which are used to define abbreviations, to provide information related to what is shown in the chart, or to provide "terminal definitions" of some sub-functions, i.e., to define sub-functions via mathematical equations or verbal descriptions. The Sequence Graph, which is the third and final part and is also optional, shows the sequence in which the processing must occur.

A Functional Development Chart may be segmented for purposes of clarity. Two types of segmentation may be used.

Lateral Segmentation is used when a chart is too large to fit on a single page. The chart is then essentially cut into pieces as a multipage map is cut into pages which

match at the page boundaries. This is illustrated in Fig. 3a.

Overlay Segmentation is used when a chart has many crossing arcs which make it difficult to read or draw. The various segments are made as if they were transparencies for an overhead projector such that if they were all overlaid upon one another the projected image would be the complete chart. Since the segments are not actually overlaid in this way, it is not essential that the same element appear in the same location on each slide but it is helpful to keep them in approximately the same places unless this interferes with the clarity of the presentation. This type of segmentation is shown in Fig. 3b.

4.4 Functional Development Chart Elements

The major component of the FDC is the Development Graph. The three characteristics of a program depicted in the Development Graph are the data elements, the data structure or relationships among the data elements, and information flow.

Data Elements

A data element may be a simple scalar quantity, an array, a table, or a COBOL or PL/1 type of structure. The data element may be in main memory where it is accessible to the arithmetic processing unit or it may be stored on some peripheral device from which it must be read into main memory before it can be processed. Different graphic symbols are used to denote characteristics of the data which are relevant to its processing.

The storage location of a data element is indicated by use of a circle or a square. Data which is in main memory is shown by a circle while data which is stored on a peripheral device is shown by a square. If a data element consists of components, some of which are in main memory and some on peripheral devices, the circle is used. If the designer has not yet decided whether the data at a given point will be in main memory or on peripheral device the circle is used. The circle is the more general symbol and can be used to represent data anyplace; the square is used to indicate that data is not directly accessible to the arithmetic processing unit.

The structural characteristics of a data element is indicated by the use of a single circle (square) or a double circle (square). Scalar elements or data structures which are not a set of items having identical formats are represented

by single circles (squares). Arrays, tables, or files of records having the same formats are represented by double circles (squares). Structures as used in COBOL or PL/1 are represented by single circles (squares). The single circle (square) is the more general symbol and may be used to represent those elements for which the double circle (square) may be used. The various symbols used to represent data elements are shown in Fig. 4.

Each data element is labelled. The label of a data element becomes the identifier of the data element in the program. For scalars or structures, the identifier is a simple name (generally chosen so as to have mnemonic value) subject to the restrictions of the programming language to be used. For arrays, tables, and files the name is followed by a pair of parenthesis inside of which is the number of elements, i.e., dimension information for an array, number of entries in a table or number of records in a file. If any of these quantities are unspecified, the number sign (#) is used to indicate that fact.

Sometimes data elements appear more than once serving different, but related, functions. For example, a data element may serve as both input and output, as is typical of a value which is to be updated. In such cases the nodes are labelled with the same name but with a suffix consisting of a period followed by the integer one, two, three, and so on to distinguish the different versions (update values). The initial mode is shown without a suffix.

Information Flow or Processing

The flow of information or data processing is shown by solid directed arcs. Each arc starts at a node and terminates on a node. Each node, except an input node, is the value computed by a function subprogram corresponding to an instruction of one of the nested-virtual machines. All the arcs terminating on a single node are labelled with the same name, which is the name of the function, suffixed with a period and an integer starting with one, then two, and so on. These integers indicate the positions of the arguments in the function call as shown in Fig. 5.

In general, information flow is shown from left to right and numbering is ascending from top to bottom. These conventions may be broken for the sake of clarity. When a chart is segmented, either all or none of the arcs terminating on a node should be shown on any segment. This prevents the error of showing the same function with different inputs on different segments.

There are occasions when the same invocation of the same function is shown on more than one functional development chart. In such cases the corresponding arcs should be labelled only on the one which is to be evaluated during the program, i.e., the first occurrence. The remaining charts represent the same call of the function and do not represent additional processing; they serve only to represent a complete picture. The designer documents this by not labelling these arcs.

Data Structure

The data structure, or the relationships among data elements, are shown by broken directed arcs.

The components of a data element are indicated by broken directed arcs radiating from the "parent" data element to the components as shown in Fig. 6. The broken directed arcs are labelled with the integers one, two, three, etc., to indicate the sequence of components. In general, this numbering is ascending from top to bottom but this convention may be ignored for purposes of clarity. When a chart is segmented, either all or none of the components of a data element should be shown on any segment so as to avoid confusion.

It is cumbersome, and frequently virtually impossible, to show all the components of an array, table, or file. It is for this reason that the double circle (square) notation is used. This notation is augmented by a selection notation which is indicated by the use of a box. The most common situation is iterative processing of the elements of an array. Figure 7a shows the selection of elements A(2), A(4), A(6), ..., A(100) of an array A. Another common case is the selectioon of a single element from a table based on the result of a computation. Figure 7b shows the selection of an element T(X) from a table T based on the computed value X. A shorthand alternative is shown in Fig. 7C; this notation may be used whenever a single element is to be selected. The notation in Fig. 7b contains a variant on that which has been presented; the semicircular node is associated with an indexing or selection operation. In this case it is labelled T(X) so the relationship is clear. The shorthand notation of Fig. 7C is equally clear; the dotted line represents a selection of an element from the array T and there is a flow of information along the solid arc from X to T(X) making the relationship clear. Furthermore, the solid arc is labelled with the special name SEL (for select) which eliminates any possible question. The broken arcs in a selection process are not labelled because there is always just a single such arc. The arrowheads on the broken arcs where they cross the selection boxes in Figs. 7a and 7b represent the selection operation.

A final application of the broken directed arc is the decision process, typically shown by a diamond-shaped outline in a flowchart and implementd by an IF statement in high-level languages. The decision process is characterizedd by a node with a single solid directed arc and multiple broken directed arcs terminating on it. The broken arcs are labelled with the possible values that the node at the beginning of the solid arc may have; most often these will be the integers one, two, three, etc. The node at the termination of these arcs is made equal to the value of the node at the start of the corresponding broken arc. This process is illustrated in Fig.8. The label on the solid arc is the special name DEC for decision.

The approach to the decision process using FSD is somewhat different from that taken when conventional methods are used. The situation depicted in Fig. 8 illustrates the difference. The flowchart in Fig. 8a indicates that one of two values of X will be computed if I = 1 or I = 2 and a value of Y will be computed if I = 3. This implies that the value of X is changed and Y is left unchanged if I = 1 or I = 2 and the value of Y is changed and X is left unchanged if I = 3.

In Fig. 8b the inputs are shown as A,B,X.1, and Y.1. The inputs X.1 and Y.1 are the values of X and Y, respectively, prior to the decision box in the flowchart. The values of X1, X2, and X3 are the three possible values of X after the decision and related processing while Y1 and Y2 are the two possible values of Y. The new values after the decision and related processing are X.2 and Y.2. Two occurrences of each, separated by an ID (identify) function are shown so there will not be ambiguity as to whether a dotted line is part of the decision operation or that it indicates a component of Z. The node Z represents the entire output consisting of the two components X and Y; this single output is required because a function subprogram can have only one output. Restricting all subprograms to function subprograms retains the relationship between the subprograms and virtual machine instructions which compute a single result.

A second optional component of the FDC is the notes section. It contains a series of sequentially numbered notes which define abbreviations, briefly describe the intended purpose of functions used in the chart, define by mathematical equations or textual description functions used in the chart, or provide other information such as data formats, type of data files, etc. The third major component of the FDC is the Sequence Graph. This optional component is used to specify the sequence in which data elements must be processed. When the sequence of processing operations does not matter, the

Sequence Graph may be omitted. Some or all of the nodes that appear in the Development Graph are used in the Sequence Graph together with solid directed arcs which show the sequence of processing as illustrated in Fig. 9. This graph indicates that data element D cannot be computed until A and B have been processed. Furthermore, both D and C must be completely processed before processing of E can start.

In addition to these three major components, a system of chart numbering is important in keeping track of the charts. Each chart is identified by a number of the form i.j - k.l where i,j,k, and l each represent unsigned decimal integers.

- i represents the level of the parent chart, i.e., the chart in which the function being defined first appears.
- j identifies the specific parent chart. This value is omitted if there is only one chart on level i.
- k is the level of the chart being prepared. Usually, but not always, k = i+1.
- l provides unique identification of the chart so as to differentiate it from the other charts on level k. This value is omitted if there is only one chart on level k.

A graphical index showing the family of charts as a tree structure is used to relate the charts. In addition, an alphabetical index, by function name, is used to avoid multiple definitions. This index indicates all functions invoked by each function and also contains references to all functions which invoke the given function. An illustration of the two indices is given in Fig. 10.

In order to insure that all required functions have been defined a dollar sign ($) is prefixed to each function invocation upon completion of its definition. Its definition is complete when a FDC for it is completed or when it is defined in a note. Certain frequently used functions, such as GET (input), PUT (output), ID (identity or move), SEL (select), and DEC (decision) are predefined and the dollar sign is included with the function invocation immediately. When all function invocations in a FDC are defined, the function name, which follows the chart identification number, is prefixed by a dollar sign; this indicates that the function is complete. When all the FDC names are prefixed by dollar signs the design process is complete.

The final element in a FDC is a brief description, usually a simple single sentence, of the function defined on the chart. The ability to describe the function of a module by a simple sentence is one of the tests used to determine whether a module has functional cohesion(17); incorporating such a description in the chart verifies that the function defined by the chart satisfies this criteria. The design process itself results in functional cohesion since the defined function computes one data element which, even if it is not a simple element, is a collection of functionally related elements. Nevertheless, the function description serves as a useful check and provides useful documentation. A typical SDC is shown in Fig.11.

Functions defined by FD are called functional functions, or F-functions, because they are functionally cohesive. Each F-function's name begins with a dollar sign (inserted when it is completely defined); this distinguishes it from ordinary functions defined during program implementation.

4.5 Functional Design Process

The FD process is one of the stepwise refinement starting at level 1 for the overrall software system and providing more and more detailed specificationns of the design on successive levels.

Each FDC defines a F-function which corresponds to an instruction for the virtual machine associated with the given level. The major component of the FDC, the Development Graph, is created using a three step process:

1. The data elements are specified by refining the definition of the input and output elements shown on the present chart. This refinement process is one of the specifying the components of the input and output elements, generally to one level.

   This step may involve only the transfer of information given in the system requirement specification to the Development Graph or it may require the designer to develop the data structures. The development of the data structure by the designer is probably the most critical aspect of the design process and the step to be most carefully considered.

2. The information flow is specified by drawing solid directed arcs indicating the sources of information required to determine all output quantities.

This step is quite straightforward providing the designer understands the functional requirements of the system. Difficulty in this area indicates one of two problems: either the designer does not understand the requirements or the designer has started with a poor data structure. In the former case, the designer must go back to the requirements specification which may be found to be incomplete. Whether it is or not the designer must get the information necessary to permit completion of the design.

In the latter case, the designer must analyze the situation to determine the source of the difficulty. Very often the difficulty is due to a confusion between the functional requirements and the procedural approach to the program. The designer must avoid any consideration of how processing will be accomplished (procedure); attention should be given only to the sources of information, identification of the results to be computed, and information flow (functional requirements).

If the above consideration does not eliminate the difficulty it is generally due to an incongruity between the data structure and the approach to the program design which the designer has in mind. This situation requires identification of the source of conflict followed by a change in the data structure or the design approach.

An iterative procedure of data structure specification, specification of information flow, modification of data structure, etc., is often required to generate a satisfactory design. This procedure may require considerable time and thought but the effort is well spent because this is the critical design activity. If the data structure and information flow complement each other, the basic program structure is good and a quality software product can be generated; if the data is poorly structured or does not match the program design, no amount of effort in subsequent stages can generate anything better than a clumsy and inefficient program. Once this phase is completed the remaining design steps are very simple and the programming procedures are much simpler and faster than they are when conventional methods are used.

3. The functions are identified by labelling the solid directed arcs with function names. The name are generally chosen to reflect the functional purpose of the function. A note is frequently included in the Notes Section which describes this purpose.

   This step is the bridge to the procedural aspect of software development. It provides the information necessary to refine the design on subsequent levels. Once a function is defined in this manner it can be isolated from the overall problem so that the designer can concentrate on this one part of the problem which is functionally cohesive module whose interfacing is clearly defined by the data elements which are its inputs and output.

5. Design Example

The overall design process is best presented by an example. The following example does not illustrate the full power of the method because of its simplicity (for the sake of brevity) but it does illustrate most of the key features of the method.

5.1 Specifications

A software system for monitoring patients in a hospital intensive care unit is to be designed.* A monitoring unit is located at each bed in the intensive care unit. Transducers which monitor various physiological factors, such as blood pressure, heart rate, respiration, temperature, etc., are connected to the monitor. The monitor also contains a number of dials which are used to input the patient's identification number, safe ranges for each factor being monitored (a high and low value are set for each factor), and the rate at which the patient is to be monitored.

The monitors connect to a centralized computer which is to check them at intervals according to the monitoring rate specified for each unit. The measured data for each patient is to be recorded in the patient's record and an alarm is to be illuminated at the nurse's station whenever a factor falls outside the safe range for the patient. There is a separate alarm light for each factor for each patient.

* The specifications for this example are based on those used by Myers in his text on composite design (14).

For this example it is assumed that four factors for each of up to 255 patients can be monitored. The monitoring can be set from one second to one hour in one second steps for each patient and the safe range values are set as four decimal-digit values which are internally stored as floating-point numbers. The patient identification number is a nine digit integer. In addition, each monitor has an off-on switch which is set on when a patient is being monitored and is otherwise off.

5.2 Design

The Functional Development Charts are developed in this section. Before doing this the general approach to the problem is considered. The first problem is finding a method for monitoring each patient at the appropriate rate. One approach is to have a timer in each monitor which interrupts the computer when it is to be monitored. A more economical and simpler approach is to have the program cyclically check all the patients and measure each patient's factors or skip the measurement depending on whether the monitoring interval has elapsed or not; this approach is the one used in the system presented. In order to minimize the time required to check whether a patient should be monitored or not, the monitoring interval and time of last monitoring for each bed in the intensive care unit are held in an array in main memory. The monitoring interval is the input from the monitor unit. This is input by a separate program which interrupts the monitoring program whenever this value is changed at any of the monitor units. A one-bit value which indicates whether the monitor is on or off is also stored for each bed in the same area of main memory and this is set by the same program which sets the monitoring interval. This program is rather simple and its design is not considered.

The monitoring program is developed using a sequence of FDC's. The first level in the sequence, shown in Fig. 12, is straightforward and obvious; it shows the basic data structure. The circular nodes used for both input and output indicate that neither of these data elements is purely from or to a peripheral device; this is due to the array of data used to control the monitoring which is stored internally.

The input to this program is derived from three separate sources: the internal array used for controlling the monitoring, a real-time clock whose output is denotated CLOCK, and the monitor units. The internal array is called INFILE, the value of the real time clock is assigned to T, and the monitor inputs are called MONIN. The value of T is assumed to vary from 0 to 24 X 60 X 60 - 1 = 86399 (= number of seconds

in day minus one). At the end of each day it resets to zero and increments by one each second. Note that MONIN is shown as a square node to indicate that it is located on peripheral devices. The developement of the input is shown in the second level FDC in Figure 13. At this point no development of the output is shown because that is a seperate problem.

The development of the data structure shown in Fig. 13 is the first step of the three-step chart development process. The numbering of the components is arbitrary, but it is often helpful to have the numbers agree with the sequence in which the data components are used.

The second step in the process is to draw solid directed arcs representing information flow. There is such an arc from each input component to the output because information from each of the components is required to determine the output.

The third and final step in the chart development process is the labelling of the arcs introduced in the second step. All the arcs terminating on a single node correspond to one functional operation and are therefore labelled with the same function name followed by a period and different integer suffixes starting with one. The name chosen in this case is MONITOR since the function is performing a monitoring operation. The assignment of the suffix values 1, 2, and 3 is arbitrary but it is generally a good practice to have these numbers correspond with the sequence in which the inputs are used in the function.

When the FDC for PROGRAM is complete, as it is now, a dollar sign ($) should be prefixed to its invocation in the first level FDC of Fig. 12. When that is done, all functions invoked by MAIN will have been defined and a dollar sign should be prefixed to the function title MAIN in Fig. 11. This indicates that all of the fuctions it invokes are defined. The result is the revised top level chart shown in Fig. 14.

The next step in the design process displays the cyclical checking of each patient to determine whether or not the patients factors should be measured. At this point the designer considers the data in greater detail. The internal data, INFILE, is an array with an entry for each bed in the intensive-care unit. The monitor inputs are a set of input devices, one for each bed in the unit. For purposes of design, these devices are treated as entries of an array. In the final implementation it may be necessary to provide a routine that selects the appropriate monitor device, but this need not be of concern at this point. The final consideration in the data development step is the output which must also be

considered as an array of elements, one for each bed. The selection of each bed in sequence is accomplished by an iterative loop shown by a box in the FDC. The partial FDC at the end of this first (data development) step is shown in Fig. 15. The new data elements are defined in the NOTES section of the chart.

The second step in the chart development process is that of adding the arcs which represent information flow. This step may also include the addition of internal nodes required for the computation of the chart output. In this example, the internal data and real-time clock output are compared and a decision is made as to whether the patient's factors should be measured or not. This is developed by calculating a monitoring decision value, MD, whose value is 0 if the monitor is off or it is not yet time to measure the patient's factors, or 1 if the patient's factors are to measured. This value is used to select the output value NOP (no operation) or the monitored output.

The inputs to MD are from INREC, the internal data for the patient, and T, the real-time clock output. MD is used to decide whether the output for the patient, OUT, is set to NOP or a new value computed from T and the patient's monitor input, MIN.

The third step provides the function labels: CH is the function that checks whether the patient's factors should be measured, MTR actually performs the monitoring operation if it is required, and $DEC is a predefined decision function. Note that the output of MTR is not labelled. This is because OUT(N) is assigned this value if MD = 1. The other possible value for OUT(N) is labelled NOP which is not actually a value but an indication that no operation is performed. The completed chart is shown in Fig. 16. The arrowheads at the boundary of the iteration box associated with INREC, MIN, and OUT represent the selection process by means of which the Nth elements of the arrays of NB elements are selected. The iteration box is labelled to indicate that N takes on all values from 1 to NB.

The sequence graph at the bottom of the chart indicates that MD must be computed before OUTPUT can be computed. It also indicates that INFILE and T must be available for the computation of MD and MONIN must be available for the computation of OUTPUT. Not all data elements are shown in the sequence graph; only those which are necessary to specify the order of those computations which must be done in a specific sequence are included.

At this point the development of chart 2-3 MONITOR is complete. The designer should return to chart 1-2, Fig. 13, and prefix all the invocations of MONITOR with dollar signs and, since those are all the function invocations in chart 1-2, a dollar sign should be prefixed to the chart title, PROGRAM. This indicates that the chart is finalized and all functions it invokes have been defined.

The fourth level of the program development is shown by two charts, one for the function CH and the other for the function MTR. The development process has isolated these two functional entities from each other so the designer can consider each one separately with no worry about their interaction, all of which is accounted for by the data elements which they share.

The development of the function CH which checks whether monitoring is necessary is attended to first. From chart 2-3, Fig. 16, it is found that CH has two inputs, INREC(N) and T, and one output, MD. The first step in the chart development process is the data development process. Three quantities are stored in the internal record for each patient: (1) an activity indicator, ACT, which indicates whether the monitor is off (ACT = 0) or on (ACT = 1); (2) an interval value, INT, which is the desired monitoring interval in seconds from 1 to 3600, and (3) the time of the last measurement of the patient's factors. T is the time in seconds and MD is a one bit value which is 0 if the patient is not to be monitored and 1 if the patient is to be monitored.

The algorithm for computing MD is chosen so as to take as little time as possible since it is performed each time the bed is checked. The FDC is shown in Fig. 17. The activity indicator ACT is checked first; if the monitor is off the check is complete, otherwise the time interval from the last monitoring time to the present, TI, is calculated and compared with the monitoring interval, INT, to evaluate MD.

The function TD is used to calculate the time interval TI. It is defined in the NOTES section using PL/1 like program statements rather than in a separate FDC; for this reason it is called a terminal function and its invocations are shown with dollar sign prefixes in the development graph. The function TCH is used to calculate the value of MD if ACT = 1. It is defined in the NOTES section and is therefore a terminal function whose invocations are prefixed by dollar signs. The output of TCH is not labelled since it is assigned to MD when it is computed. The other possible value of MD is the constant 0 as shown in Fig. 17.

The sequence graph shows that INREC must be available in order to start. The first node shown after that is ACT which is a component of INREC, rather than a computed value, and therefore available from the start. The reason for this is to indicate that ACT is checked before invoking TD and TCH; if ACT = 0 the other functions are not invoked at all. The computation of TI and final evaluation of MD shown in the sequence graph occur only if ACT = 1.

The chart title CH is prefixed by a dollar sign because all the functions it invokes are shown with dollar sign prefixes. At this point the designer turns back to chart 2-3, Fig. 16, and prefixes the two occurrences of CH by dollar signs. In order to completely finalize chart 2-3 it is necessary to define the function MTR.

The definition of MTR is shown in Chart 3-4.2 in Fig. 18. This function measures and processes the patient's factors. Its inputs are T and MIN(N) and its output is OUT(N) as can be seen in Chart 2-3, Fig. 16.

The first step in developing the chart is the data development step. T is a simple scalar quantity. MIN is the input from the monitor device. The first requirement is to transfer MIN into main memory; this is done using the predefined function $GET. This data consists of three major components: the patient identification number, PN, set on monitor dials; the factors, FRS, measured by the transducers connected between the patient and monitoring unit; and the safe ranges on these factors, RNGS, set on monitor dials. The factors must be converted from analog to digital form; this may be done by individual analog-to-digital converters in the monitor units or by a single unit at the computer input. This detail is not of consequence at this point in the design. The output data for the patient consists of three main components: (1) an updated value of the monitoring time, TM, contained in INREC; (2) recorded values of the measured factors and the time of their measurement in the patients file, PFIL; and (3) the alarm output at the nurses' desk, ALM. The updated monitoring time is labelled TM.1 to differentitate it from TM which is part of INREC while at the same time indicating that both identifiers reference the same value. This is handled by a COMMON block in FORTRAN or a DEFINED attribute in PL/1. In a similar manner, the patients file is labelled PFIL.1. The original PFIL is not shown in any of the FDC's but is implicit. The reason for this is that OUT(N), and all of its components, can be any of 255 different data elements whereas the patients file is a single file with subfiles for all of the patients. The changing of patients in the intensive care unit prevents a fixed relationship between bed number and

patient number. As a result, the patient number must be dialed into the monitor unit and this must be used to select the appropriate patient subfile. In order to have all of the patient file references access the same patient file, they are each labelled PFIL.1 and all of these are caused to reference the same value as PFIL.

The function $MOV is predefined and simple. It causes the value of its input to be assigned to its output. The function SRC selects the proper patient subfile using PN and records the time and measured factors. The function ALARM checks each factor to determine whether it falls inside or outside of the safe range and outputs an alarm signal if it is outside the range. Each alarm signal activates a different lamp on a panel at the nurses' desk. The predefined function $PUT transfers data to a peripheral device.

The development of the SRC function is shown in Chart 4.2-5.1, Fig 19. The value of PN is used to select the proper file PFIL which is an array of subfiles whose dimension is not specified and is therefore indicated by the number sign (#). The predefined function $ID implies that its input and output reference the same values; in this case the same identifier is used and the function $ID is used to show that the input element is used for indexing. In other cases $ID may be used to cause a COMMON (FORTRAN) or DEFINED (PL/1) type of relationship to be set up between identifiers. The function RCD is developed in Fig. 20. In this chart $ID is used to cause TR to reference T and FR to reference FRS. The identifiers T and FRS cannot be used directly because they are components of data elements other than PREC (PN) and the same identifier used in different structures do not reference the same values.

The one function remaining to be defined is ALARM which is invoked in Chart 3-4.2, Fig. 18. The development of this function is shown in Chart 4.2-5.2, Fig. 21. In Fig. 18 the inputs are seen to be FRS and RNGS and the output is ALM. In order to compare the factors and ranges it is necessary to select each factor in succession and check it against the range values for it. This is shown by the iteration box in Fig. 21. The NOTES section defines the new identifiers. The function FA performs the comparison of a given factor with the range values. Its development is shown in Chart 5.2-6.2, Fig. 22. In this chart the range data element, RIN, is developed as a high value, RH, and a low value, RL. The function CA which computes the alarm is a terminal function, i.e., it is defined in a note in the NOTES section of the chart.

The Functional Design of the patient monitoring system is complete. All the invoked and defined function names would be prefixed by dollar signs at this point if the designer attended to this function upon completion of each FDC. A review of the charts shows the stepwise refine- ment of the design and the segmentation of the program into distinct functional modules.

The presentation of a completed work cannot show the thinking that went into it. The data structures used here seem the most obvious ones to use but this is only because the design is complete and everything fits. The data structure is not as obvious when one sits down to tackle a new problem. Some appreciation of this may be had by considering the output of the program. The most obvious data structure to use is the physical data structure. For this program that would imply the structure shown in Fig. 23. The structure shown in Fig. 23 would appear over a number of levels of FDC's if it were used as the basis for the program design; it is different from but no more complex than the one used. The problem with using it is that the program design will not work out neatly. The designer's creativity becomes important in developing a suitable data structure which can be overlaid on the existing physical data structure and which permits the development of a well structured program. The Functional Design method is a great aid in this process because (1) it provides a means of breaking a large problem into a series of smaller steps, (2) it isolates functional modules so that the designer's attention can be concentrated on one part of the problem at a time, and (3) it provides a visual picture of the design which permits the designer to easily sense whether the design is good or not. A mess of crossing lines is an immediate signal that there is a problem. If, on the other hand, the designer ends up with a series of relatively simple, easy to follow FDC's then the design is undoubtedly a good one. All that remains then is implementation of the design as a program.

6. Functional Programming

It is possible to write a program from a set of FDC's using existing languages but many of the benefits of the method are lost. Specifically, it is not possible to write a subprogram for each FDC and combine these into a program. The reason for this is the lack of a facility in existing languages for dealing with the data structures represented by the broken directed arcs in the FDC's.

The Functional Programming Language (FPL) is designed to provide this capability. This language has not yet been implemented nor has a processor been developed for it. At

this point the general requirements have been formulted and an algorithm for processing it has been worked out in sufficient detail to demonstrate the feasibility of using such a language. With the use of such a language it is possible to write a subprogram, called a functional-function, for each FDC using very simple procedures which should result in programs with high correctness quality scores. The programming procedures are sufficiently straightforward that it should be possible to generate much of the code by computer. A very important advantage of Functional Programming is that intermodule linkages are automatically taken care of thus eliminating a major source of errors in programming.

6.1 Functional Programming Language Elements

The Functional Programming Language (FPL) consists of three main elements: (1) a basis language; (2) a data description facility; and (3) functional-functions and associated argument notation.

Basis Language

The basic language is an existing high-level compiler language. PL/1 appears to be the best choice in terms of providing the facilities required for Functional Programming but other languages could be used instead.

Data Description Facility

The Data Description (DAD) is a statement in FPL which provides a comprehensive description of all the data involved in a program whether internal or external to main memory. It contains the attributes of the data as well as the relationships among the data elements as defined by the FDC's. The DAD statement is similar to a combination of a PL/1 DECLARE statement and a COBOL DATA DIVISION.

Functional-functions and Argument Notation

Functional functions (F-functions) are functions which are derived from the FDC's. They differ from ordinary functions by the way in which arguments are handled. An argument notation, similar to that used in the Xerox Sigma series Metalanguage macro system (18), is used in writing the definitions of F-functions.

The problem in handling the arguments of F-function is that it is necessary to reference components of the arguments. This is solved by referencing each argument by its ordinal position and using a notation similar to the qualification

notation used with PL/1 structures to reference the components.

6.2 Writing a Functional Program

The FPL syntax and semantics is not presented in detail because the language is not yet fully defined. Instead, a program for the patient monitoring system designed in section 5 is written with commentary on the program and its preparation. The language used is an augmented PL/1 and should be understandable to most readers with knowledge of a high-level language. For those unfamiliar with PL/1 the following notes will be of help in following the programs:

1. It is a free format language; statements can start and end anywhere.

2. Statements are terminated by semicolons (;).

3. Identifiers begin with a letter of the extended alphabet (A-Z$#@) and consist of any number of these letters,the digits 0-9, and the underline (_) character.

4. Statements may be identified by a label which is any identifier followed by a colon (:) to separate it from the remainder of the statement.

5. Comments are bracketed by the symbols /* and */. They may be inserted anyplace that a blank is permitted.

Main Program

The main program begins with a procedure statement which specifies the name of the program (MAIN) and the fact that it is a main program; this statement is a standard PL/1 statement. This is followed by a comment which describe the program. The Data Description statement follows. This statement is quite lengthy and normally accounts for the major part of the program. The next statement is the statement which performs all the work of the program; it is directly derived from the top-level FDC The last statement is an END statement which identifies the end of the program.

The DAD statement includes the attributes of all the identifiers used in the program, except those which are local to F-functions, and the formats of external values. The DAD also represents the structure of the data elements. The DAD statement required to represent the data structure of Fig. 24a is shown in Fig. 24b. The DAD statement in Fig. 24 is incomplete in that the attributes are not included; it shows

only the structure. Normally, the structure shown in Fig. 24a would require three levels of FDC charts.

The program listing for the main program is shown below. It is necessary to trace through all the FDC's to follow the DAD statement. Some of the attributes of the identifiers may not be clear to those not familiar with PL/1 but they are not very important for purposes of this paper. Not all the information required to determine all the attributes and formats was given previously but they should be understandable to one versed in PL/1.

```
$MAIN:   PROC OPTIONS (MAIN);
         /* PATIENT MONITORING SYSTEM */
         DAD
         1.1  INPUT,
              2.1  INFILE INT,
                   3.1  INREC(NB),
                        4.1  ACT FIXED BIN (1) INIT (0),
                        4.2  INT FIXED BIN (12),
                        4.3  TM FIXED BIN (17) INIT (86399),
              2.2  T INT DEF (CLOCK),
              2.3  MONIN EXT,
                   3.1  MIN(NB) FILE (MONITOR),
                        4.1  PN FIXED DEC (9),
                        4.2  FRS,
                             5.1  FIN(NF) FLOAT BIN (24),
                        4.3  RNGS,
                             5.1  RIN(NF),
                                  6.1 RH FLOAT BIN (24),
         1.2  OUTPUT,
              2.1  OUT(NB),
                   3.1  TM.1 DEF TM,
                   3.2  PFIL.1 DEF PFIL,
                   3.3  ALM FILE (ALARM)
                        FORMAT ((4)B(1)),
         1.3  PFIL EXT,
              2.1  PREC (#) FILE PATFILE
                   FORMAT (F(5),(4)F(6)),
                   3.1  TR DEF T,
                   3.2  FR DEF FRS,
         1.4  NB FIXED BIN (8) INIT (255),
         1.5  N FIXED BIN (8),
         1.6  TI FIXED BIN (18),
         1.7  NF FIXED BIN (3) INIT (4),
         1.8  M FIXED BIN (3);
         OUTPUT = $PROGRAM (INPUT);
         END $MAIN;
```

The statement

```
OUTPUT = $PROGRAM (INPUT);
```

is a direct translation of the Development Graph of chart 0-1. The DAD statement contains all of the following information: (1) data structure; (2) attributes of internal values; (3) formats of external values; and (4) files from and to which input and output are transferred.

Differences between standard PL/1 and FPL as used in this example are:

(1) There is no DAD in PL/1. Elements of the DECLARE statement and FORMAT lists are combined in the DAD together with various other items, e.g., the FILE term.

(2) Identifiers followed by a period and then a decimal integer are used in FPL but not in PL/1.

(3) The number sign for an unspecified array dimension is not part of PL/1.

(4) The EXT and INT attributes are used differently in PL/1.

(5) Variable array dimensions are not used in a PL/1 main program.

Functional-functions

The remainder of the program is implemented by a set of F-functions, one for each FDC and one for each terminal function. Each F-function starts with a procedure statement similar in format and function to the procedure statement for the main program . The differences are the change in function name and the OPTIONS(FFUN) which causes it not to be a standard PL/1 statement. The next statement is a comment statement containing the function's description from the FDC. The next statements are the statements which implement the program and the last statement is an END statement.

The statements which implement the program use the argument notation. This notation uses integer values preceded by a dollar sign to represent each of the function arguments and zero preceded by a dollar sign to represent the function output. The first input argument is represented by the integer one, the second by two, and so on. The order of the arguments is set by the integer suffixes appended to the

function name when it is invoked. An example is shown in Fig. 25. Part of a FDC is shown in Fig. 25a while the argument notation is listed in Fig. 25b.

The components of an argument are referenced by an extension of this notation. An argument component is specified by a dollar sign, the argument integer, as described above, followed by a period and a integer specifying the component as shown on the dotted line connecting the component to the argument. An example of this is shown in Fig. 26a which is the next level for the chart segment shown in Fig. 25a. The operator notation is listed in Fig. 26b.

The program listings for the F-function definitions are given below using the argument notation described above. The handling of DO loops is the usual PL/1 technique; the loop starts with a DO statement and is terminated with an END statement. Decision operations are handled using standard PL/1 IF-THEN-ELSE statements.

```
$PROGRAM: PROC OPTIONS (FFUN);
          /* MONITORS INTENSIVE-CARE PATIENTS */
          $0=$MONITOR ($1.1, $1.2, $1.3);
          END $PROGRAM;

$MONITOR: PROC OPTIONS (FFUN);
          /* CONTROLS PATIENT MONITORING */
          DAD MD FIXED BIN (1);
          DO N = 1 TO NB;
          MD = $CH($1.1(N),$2);
          IF MD = 1 THEN DO;
              $0.1(N) = $MTR($2,$3.1(N));
              END;
          END;
          END $MONITOR;

     $CH: PROC OPTIONS (FFUN);
          /* CHECKS MONITOR PERIOD */
          IF $1.1 = 0 THEN $0 =0;
             ELSE DO;
                  TI = $TD ($2,$1.3);
                  $0 = $TCH (TI, $1.2);
                  END;
          END $CH;

     $TD: PROC OPTIONS (FFUN);
          /* COMPUTES TIME DURATION */
          $0 = $1 - $2;
          IF $0 <= 0 THEN $0 = $0 + 86400;
          END $TD;
```

```
$TCH: PROC OPTIONS (FFUN);
      /* CHECKS TIME */
      IF $1 >= $2 THEN $0 = 1;
                  ELSE $0 = 0;
      END $TCH;

 $MTR: PROC OPTIONS (FFUN);
      /* MONITORS PATIENT */
      $GET ($2);
      $0.1 = $MOV ($1);
      $0.2 = $SRC ($2.1, $1, $2.2);
      $0.3 = $ALARM($2.2, $2.3);
      $PUT ($0.3);
      END $MTR;

 $SRC: PROC OPTIONS (FFUN);
      /*  SELECTS PATIENT SUBFILE AND RECORDS
          FACTORS */
      DAD
      1  POUT,
                    2.1  TR DEF T,
                    2.2  FR DEF FRS;
      POUT = $RCD ($2,$3);
      CALL OUTPUT ($1,POUT);
      END $SRC;

 $RCD: PROC OPTIONS (FFUN);
      /* RECORDS FACTORS AND TIME */
      END $RCD;

$ALARM: PROC OPTIONS (FFUN);
      /* COMPUTES PATIENT ALARM OUTPUT */
      DO M = 1 TO NF;
      $0.1(M) = $FA($1.1(M), $2.1(M));
      END;
      END $ALARM;

  $FA: PROC OPTIONS (FFUN);
      /* COMPUTES FACTOR ALARM */
      $0 = $CA($1,$2.1,$2.2);
      END $FA;

  $CA: PROC OPTIONS (FFUN);
      /* COMPUTES ALARM */
      IF $2.3 <= $1 AND $1 <= $2
         THEN $0=0; ELSE $0=1;
      END $CA;
```

These F-functions complete the program implementation. The definition of $RCD is incomplete because it depends on the nature of the peripheral device and the file structure which have not been specified. The separate functions make program maintenance very simple because most changes require modification of a limited number of functions without affecting the remaining functions.

The brevity of the functions and their direct relationship to the FCD's makes the programming process a very simple one. Its simplicity makes it less prone to errors than conventional programming is.

7. Functional Language Processor

A Fuctional Language Processor (FLP) is necessary to process programs written in FPL. The FLP may be implemented as a compiler which translates FPL source language into machine language. This tends to be the most efficient technique but it is also the most expensive approach. A somewhat less efficient but simpler and less costly method is to use a preprocessor which translates FPL into the basis language which is then translated into machine language by an existing compiler. The latter approach is the one used here.

The major function of the preprocessor is to combine the main program and F-functions into a single program with the argument notation replaced by the appropriate variables. The processor is able to make these substitutions by referencing the DAD statement. The processor must also substitute standard PL/1 input and output statements for the FPL $GET and $PUT statements and it must generate DECLARE statements from the DAD statement.

The processor replaces the argument notation by referring back to the DAD statement to find the component of the arguments. The arguments of each function processed are actual arguments determined by processing the function at the preceding level. Using the format and file information in the DAD statement the procesor is able to generate the required input and output statements. The program generated for this example uses some non-standard file notation which is a result of the program requirements, not the use of FSD. The DECLARE statement is also generated from data in the DAD statement.

Identifiers containing a period and integer suffix are not valid PL/1 identifiers. The processor replaces the period by a dollar sign thus creating valid PL/1 identifiers. To avoid problems, it is necessary that no FPL identifiers end with a dollar sign followed by integers.

The processed program showing F-function invocations, marked with plus (+) sign and having no affect on program operation, is shown below.

```
$MAIN: PROC OPTIONS (MAIN);
       DCL N FIXED BIN (8), NB FIXED BIN (8) INIT (255), MD
           FIXED BIN (1) TI FIXED BIN (18), T FIXED BIN (17)
           DEF CLOCK, M FIXED BIN (3), NF FIXED BIN (3)
           INIT (4),
           1 INREC (255),
            2 ACT FIXED BIN (1) INIT (0),
            2 INT FIXED BIN (12),
            2 TM FIXED BIN (17),
           1 MIN (255),
             2 PN FIXED DEC (9),
             2 FRS,
               3 FIN (4) FLOAT BIN (24),
             2 RNGS,
               3 RIN (4),
                 4 RH FLOAT BIN (24),
                 4 RL FLOAT BIN (24),
           1 OUT (255),
             2 TM$1 FIXED BIN (17) DEF TM,
             2 ALM,
               3 A(4) FIXED BIN (1),
           1 POUT,
              2 TR DEF T,
              2 FR DEF FRS;
     +OUTPUT = $PROGRAM (INPUT);
     +OUTPUT = $MONITOR (INFILE, T, MONIN);
        DO N = 1 TO NB;
        +MD = $CH (INREC(N), T)
        IF INREC(N).ACT = 0 THEN MD = 0;
           ELSE DO;
                +TI=$TD (T,INREC(N).TM);
                TI=T-INREC(N).TM;
                IF TI <= 0 THEN TI = TI + 86400;
                +MD=$TCH (TI,INREC(N).INT);
                IF TI >= INREC(N).INT THEN MD = 1;
                ELSE MD = 0;
                END;
        IF MD = 1 THEN DO;
                     +OUT(N) =$MTR (T,MIN(N));
                     GET FILE (MONITOR(N)) EDIT (MIN(N)) (F(9),
                     (12)F(6)); OUT(N).TM$1 = T;
                     +OUT(N).PFIL$1 = $SRC (MIN(N).PN, T,
                     MIN(N).FRS);
                     +POUT = $RCD (T, MIN(N).FRS);
                     CALL OUTPUT (MIN(N).PN,POUT);
                     +OUT(N).ALM = $ALARM (MIN(N).FRS,
```

```
                    MIN(N).RNGS);
                         DO M = 1 TO NF;
                         +OUT(N).ALM.A(M) = $FA
                                   (MIN(N).FRS.FIN(M),
                                   MIN(N).RNGS.RIN(M));
                         +OUT(N).ALM.A(M) = $CA
                                  (MIN(N).FRS.FIN(M),
                                  MIN(N).RNGS.RIN(M).RH,
                                  MIN(N),RNGS.RIN(M).RL);
                         IF MIN(N).RNGS.RIN(M).RL
                            <= MIN(N).FRS.FIN(M)
                            AND MIN(N).FRS.FIN(M)
                            <= MIN(N).RNGS.RIN(M).RH
                                  THEN OUT(N).ALM.A(M) = 0;
                                  ELSE OUT(N).ALM.A(M) = 1;
                         END;
                    PUT FILE(ALARM) EDIT(OUT(N).ALM) ((4)B(1));
                    END;
          END;
          END $MAIN;
```

The PL/1 program, with the F-functions invocations eliminated, follows:

```
$MAIN:  PROC OPTIONS (MAIN);
        DCL N FIXED BIN(8), NB FIXED BIN (8) INIT (255), MD
            FIXED BIN(1), TI FIXED BIN (18), T FIXED BIN (17)
            DEF CLOCK, M FIXED BIN (3), NF FIXED BIN (3)
            INIT (4),
            1 INREC(255),
              2 ACT FIXED BIN (1) INIT (0),
              2 INT FIXED BIN (12),
              2 TM FIXED BIN (17),
            1 MIN (255),
              2 PN FIXED DEC (9),
              2 FRS,
                3 FIN (4) FLOAT BIN (24),
                  4 RHH FLOAT BIN (24),
                  4 RL FLOAT BIN (24),
            1 OUT (255),
              2 TM$1 FIXED BIN (17) DEF TM,
              2 ALM,
                3 A (4) FIXED BIN (1),
            1 POUT,
              2 TR DEF T,
              2 FR DEF FRS;
              DO N = 1 TO NB;
              IF INREC(N).ACT = 0 THEN MD = 0;
                 ELSE DO;
                      TI = T - INREC(N).TM;
```

```
                IF TI <= 0 THEN TI = TI + 86400;
                IF TI > INREC(N).INT THEN MD = 1;
                ELSE MD = 0;
                END;
          IF MD = 1 THEN DO;
                      GET FILE (MONITOR(N)) EDIT
                      (MIN(N))(F(9),(12)F(6));
                      OUT(N).TM$1 = T;
                      CALL OUTPUT (MIN(N).FRS,
                      MIN(N).RNGS);
                          DO M = 1 TO NF;
                          IF MIN(N).RNGS.RIN(M).RL <=
                             MIN(N).FRS.FIN(M) AND
                             MIN(N).FRS.FIN(M) <=
                             MIN(N).RNGS.RIN(M).RH
                             THEN OUT(N).ALM.A(M) = 0;
                             ELSE OUT(N).ALM.A(M) = 1;
                           END;
                       PUT FILE (ALARM) EDIT
                       (OUT(N).ALM)((4)B(1));
                       END;

     END;
     END $MAIN;
```

The final program is a single PL/1 program using one subroutine for outputting the patient data. This subroutine must find the proper storage area on the peripheral device, most likely a disk, and place the data after the previously stored data. The details of this subroutine are not presented in this paper.

The elimination of the F-functions by the processor results in a short and simple program with no overhead for function invocation. Statements can be tagged to indicate the F-fuctions from which they come; this could be quite useful for debugging purposes.

Maintenance is done on the initial source program which must then be reprocessed before execution. Much of the improved maintainability associated with FSD is not present in the processed program.

The relative simplicity of the final program is result of the significant design effort expended on the program using FSD.

8. Conclusions

The use of FSD reduces the software design process to one of finding or designing a suitable conceptual data structure. Once this is done the rest of the design process is quite simple. The graphical approach facilitates design by humans because there is an appeal to the human ability to grasp the gestalt, or overall pattern, of the problem.

Among the advantages of FSD are the following:

(1) FSD provides the designer with a design procedure which concentrates the designer's efforts on the development of a conceptual data structure. The balance of the design procedure is quite straightforward and simple.

(2) The source program designed using FSD consists of a set of modules each of which has, according to Constantine's definitions, functional cohesiveness which is the highest and most desirable level of binding of the elements within a module.

(3) The coupling of the modules is very loose, another highly desirable attribute according to Constantine. Moreover, the linkages between modules are all contained in the data structure and are automatically taken care of by the Functional Language Processor. This eliminatess a major source of errors in the programming and design processes.

(4) The Functional Development Charts provide quantitative measures, i.e., number of nodes, number of branches, number of charts, etc., of program complexity which can be used to estimate programming effort and cost.

(5) The definition of functions as instructions for nested virtual machines increases code reusability which results in enhanced programming efficiency and reliability.

(6) The simplicity of the graphics coupled with the control mechanisms for checking completeness facilitates the use of computerized support in the tasks of verification and validation, librarianship, and automatic programming.

(7) The method has the general advantages associatded with top-down methods such as simplified testing and suitability for chief programmer team management.

Perhaps the most significant feature of FSD is the vast improvement in software maintainability obtained with its use. The structure of a program as a set of nested, functionally-cohesive, loosely-coupled modules make it possible to alter the single module which implements a

function which is to be changed. The documentation, in the form of Functional Development Charts and source listings of the F-functions, makes it quite simple to locate the precise point which must be changed. The isolation of data structure and format specifications in Data Description statements, the independence of the F-functions, and the automatic processing of module linkages make implementation of the change very simple and economical.

The Functional Design method is fairly well developed. While application of the method to new problems will undoubtedly uncover areas for improvement, there is at this time a comprehensive documented procedure for the design phase. The major area for additional work is that of data structures; an attempt to find general structures or methods for deriving structures which can be applied to a wide variety of problem types.

The development of the Functional Programming Language and a processor for it is the next major project required to complete the Functional Software Development methodology. The feasibility of these components has been demonstrated but a substantial effort is necessary for their implementation.

References

1. Baker, F.T. "Chief Programmer Team Management of Production Programming", IBM Systems Journal, V 11 n 1, January 1972.

2. Bates, D. (Ed) Structured Programming, Infotech State of the Art Report, Berkshire, England, 1976.

3. Beam, W. R. "Microprocessors and Defense Systems", AGARD Lecture Series No.87, Microprocessors and Their Applications, Griffiss AFB NY 13441, 14 April 1977.

4. Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., McLeod, G. S., Merritt, N. J. "Characteristics of Software Quality", Doc. TRW 25201-6001-RU-00, NBS Contract #3-36012, 28 December 1973.

5. Caine, S. H. and Gordon, E. K. "PDL - A Tool for Software Design", National Computer Conference, 1975.

6. Dijkstra, E. W. "Goto Statement Considered Harmful", Communications of the ACM, V 11, March 1968.

7. Drossman, M. M. "Software Quality", Technical Report submitted to Rome Air Development Center, May 1977.

8. Eiden, H. J. and Moore, C. R. User Requirements Language (URL) Users Manual, Information Systems Technology Applications Office, Hanscom AFB, MA 01731, 1975.

9. Hamilton, M. and Zeldin, S. "Integrated Software Development System/Higher Order Software Conceptual Description (Version I)", Research and Devlopment Technical Report ECOM-76-0329-F, November 1976.

10. Jackson, M. "Data Structure as a Basis for Program Design", Structured Programming, Infotech State of the Art Report, Berkshire, England, 1976.

11. Liskov, B. H. and Zilles, S. N. "Specification Techniques for Data Abstraction" IEEE Transactions on Software Engineering VSE-1 n 1, March 1975.

12. McCall, J. A., Richards, P. K., Walters, G. F. "Factors in Software Quality", Preliminary Interim Technical Report No. 1, RADC Contract No. F030602-76-C-0417, October 1976.

13. McCall, J. A., Richards, P. K., Walters, G. F. "Factors in Software Quality", Preliminary Interim Technical Report No. 2, RADC Contract No. F030602-76-C-0417, January 1977.

14. Myers, G. j. Reliable Software Through Composite Design, First Edition, Petrocelli Charter, New York, 1975.

15. SofTech, Inc. An Introduction to SADT Structured Analysis and Design Technique, 9022-78R, SofTech, Inc., Waltham, MA 02154, November 1976.

16. Stay, J. F. "HIPO and Integrated Program Design", IBM Systems Journal, V 15 n 2, 1976.

17. Stevens, W. P., Myers, G. J., Constantine, L. L., "Structured Design", IBM Systems Journal, V 13 n 2, 1974.

18. Xerox Meta-Symbol, Sigma 5-9 Computers, Language and Operations Reference Manual, 90-09-52E, Xerox Data Systems, El Segundo, CA, July 1971.

19. Yourdon, E. Techniques of Program Structure and Design, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1975.

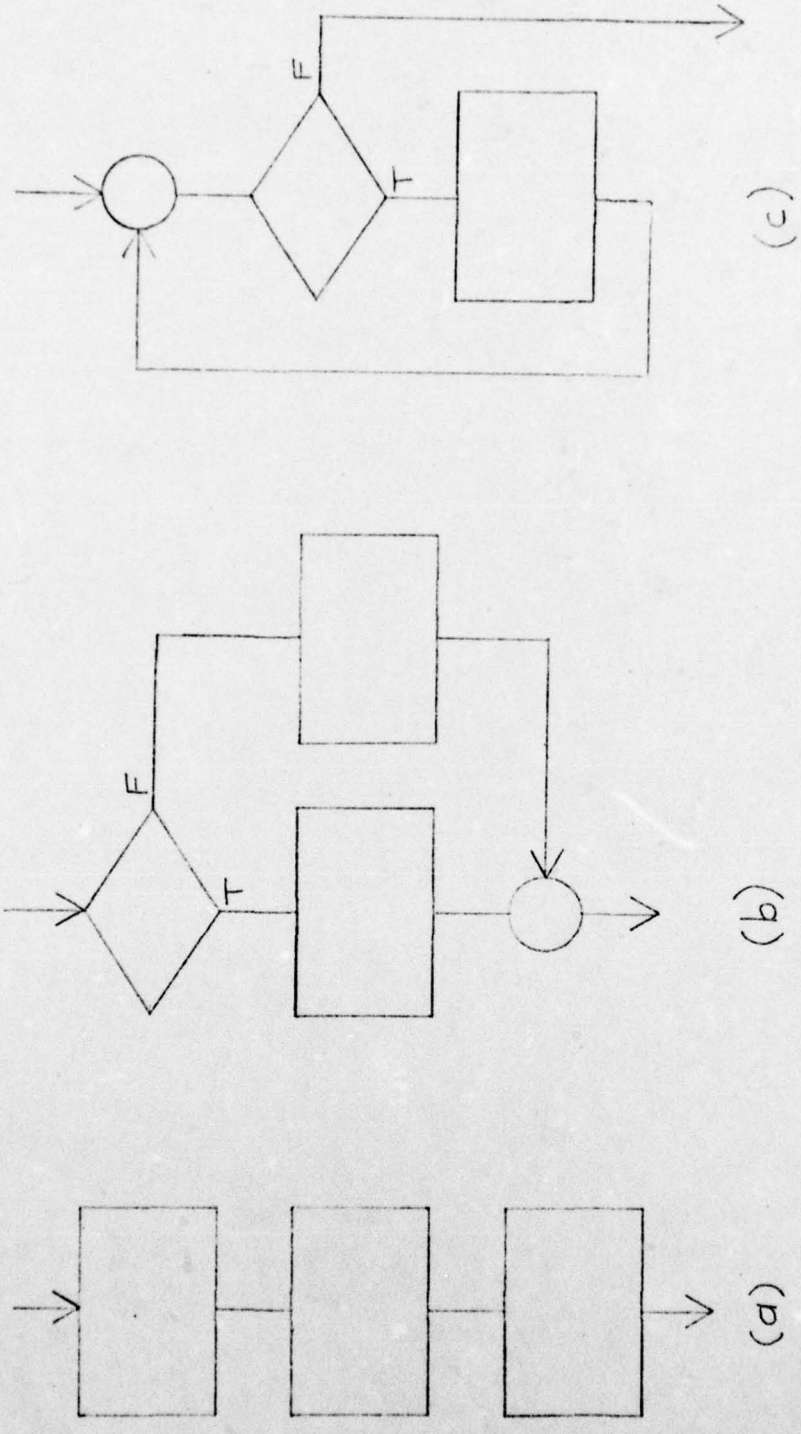


Figure 1. Structured programming constructs:
(a) sequence; (b) IF-THEN-ELSE; (c) DO-WHILE

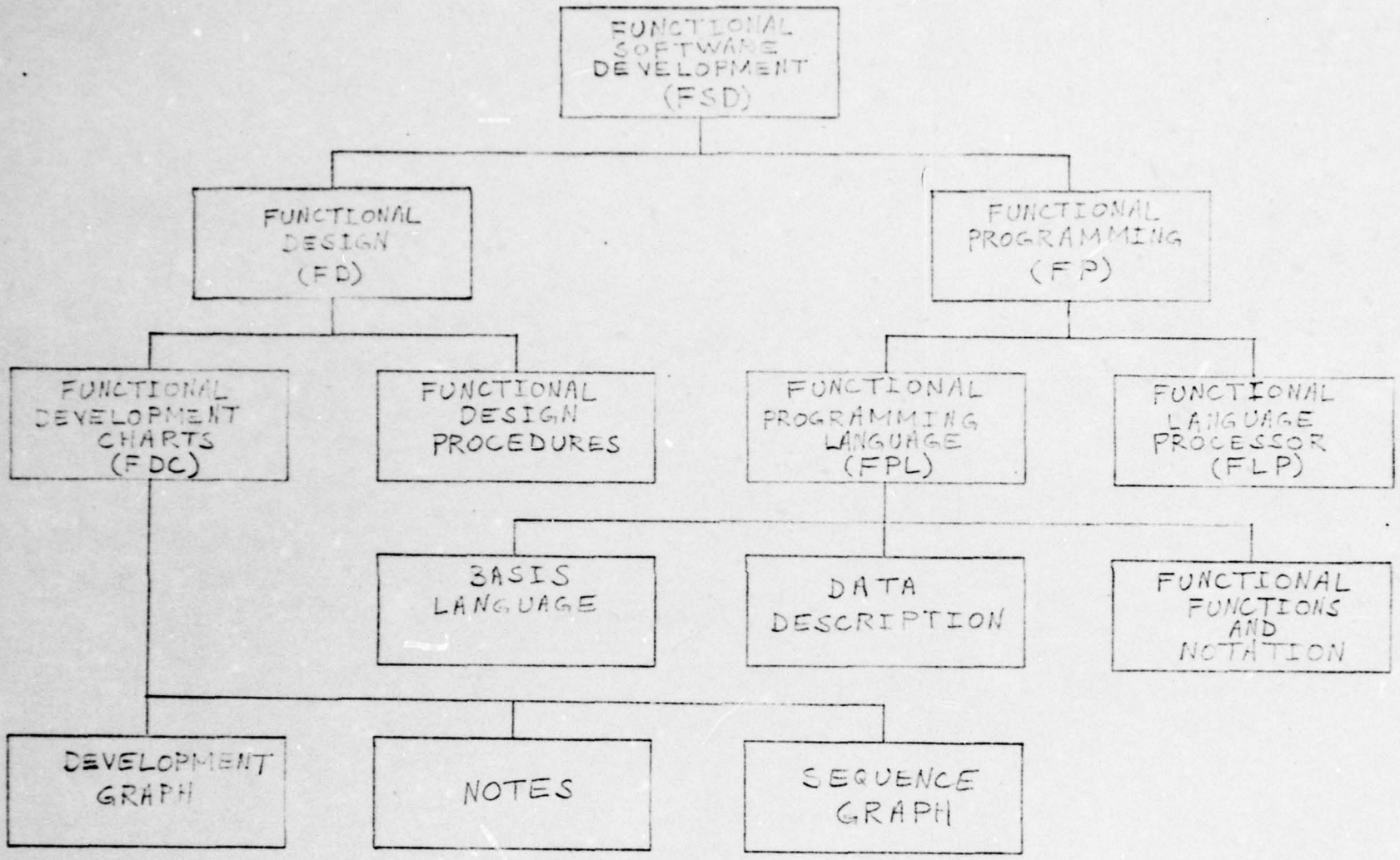
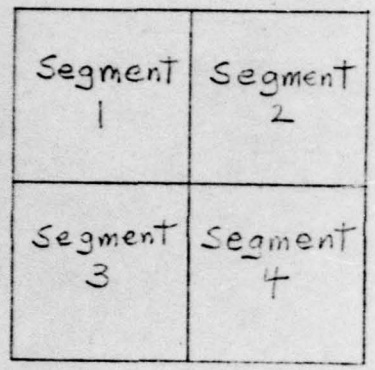
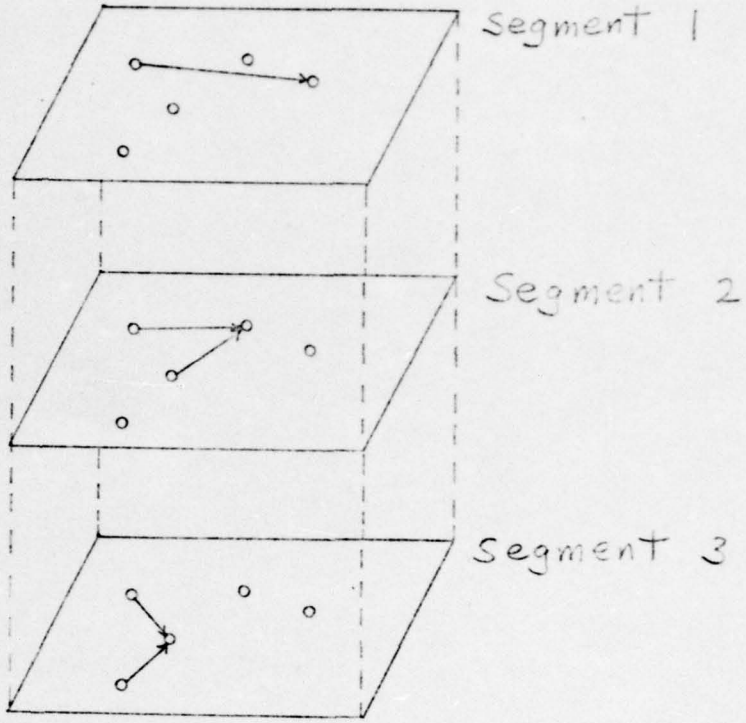


Figure 2. Functional Software Development Components



(a)



(b)

Figure 3. Segmentation (a) Lateral; (b) overlay

Type of Data	Primary Symbol	Optional Symbols
Internal scalar or structure	○	
External scalar or structure	□	○
Internal array, table, or file	◎	○
External array, table, or file	▣	□ ◎ ○

Figure 4. Symbols for representation of data elements

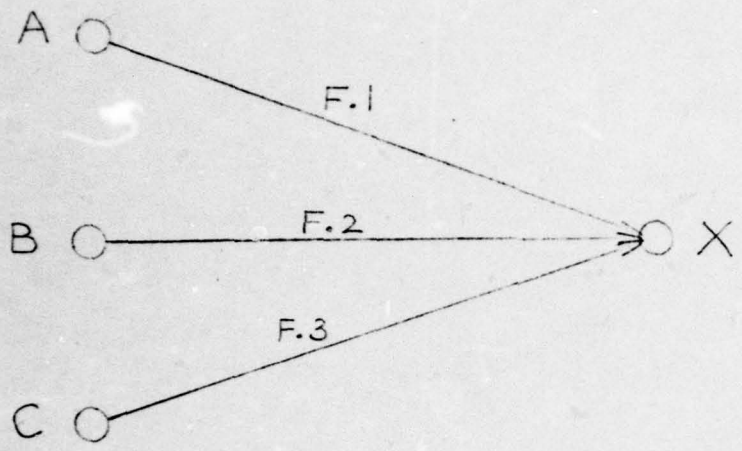


Figure 5. Function representation

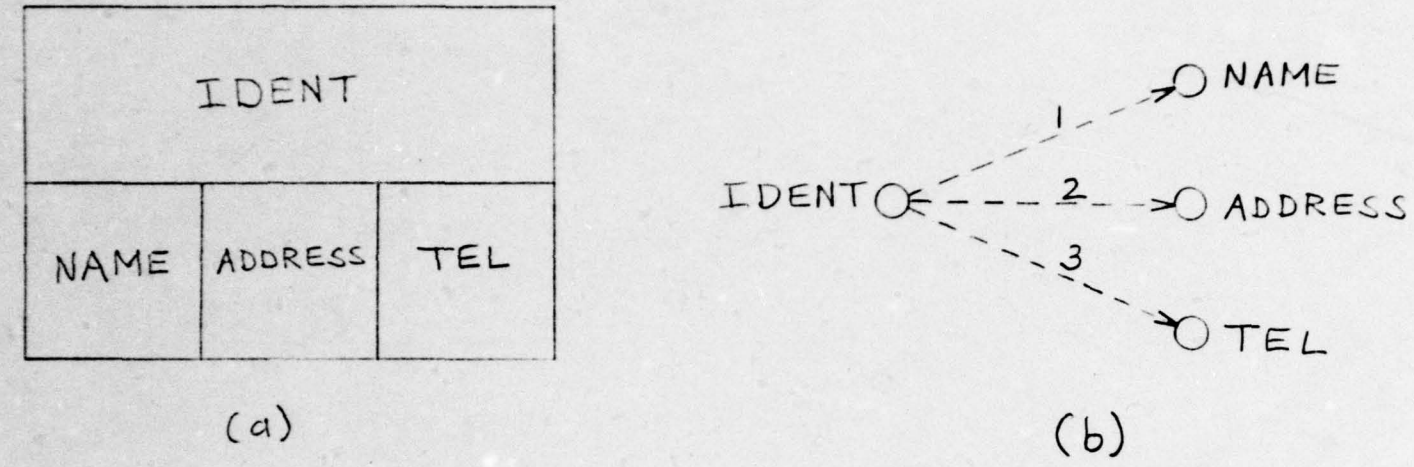


Figure 6. Data structures. (a) Data and Components; (b) development graph representation

Figure 7. Data selection . (a) Iterative selection; (b) selection of single element; (c) shorthand notation for single element selection

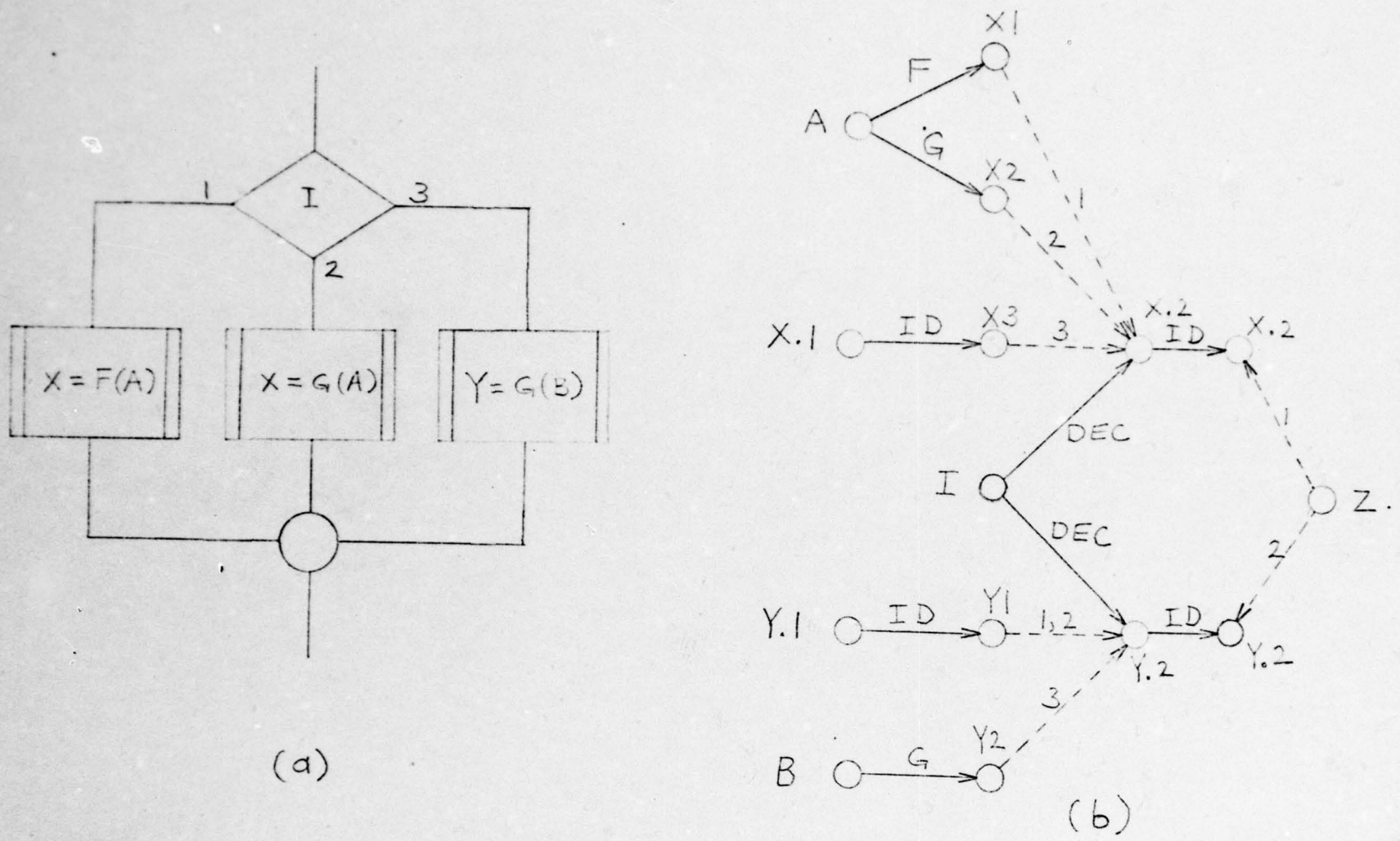


Figure 8. Decision process. (a) Flowchart representation; (b) development graph representation

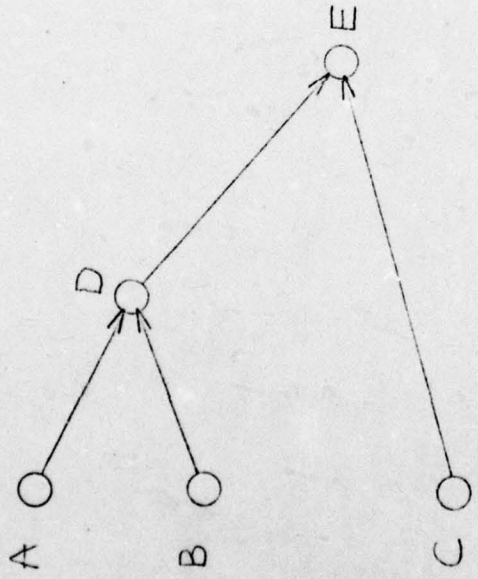
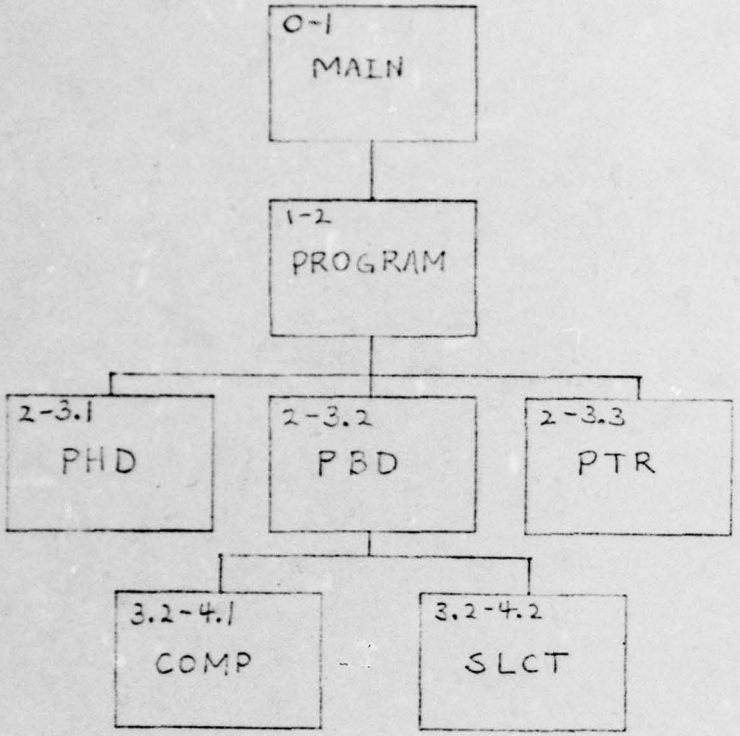


Figure 9. Sequence Graph



(a)

FUNCTION	INVOKES	REFERENCED BY
COMP(3.2-4.1)	———	PBD (2-3.2)
MAIN (0-1)	PROGRAM(1-2)	———
PBD(2-3.2)	COMP(3.2-4.1) SLCT(3.2-4.2)	PROGRAM(1-2)
PHD(2-3.1)	———	PROGRAM(1-2)
PROGRAM(1-2)	PBD(2-3.2) PHD(2-3.1) PTR(2-3.3)	MAIN(0-1)
PTR(2-3.3)	———	PROGRAM(1-2)
SLCT(3.2-4.2)	———	PBD(2-3.2)

(b)

Figure 10. Functional Development Chart indices. (a) Graphical index; (b) Alphabetical index.

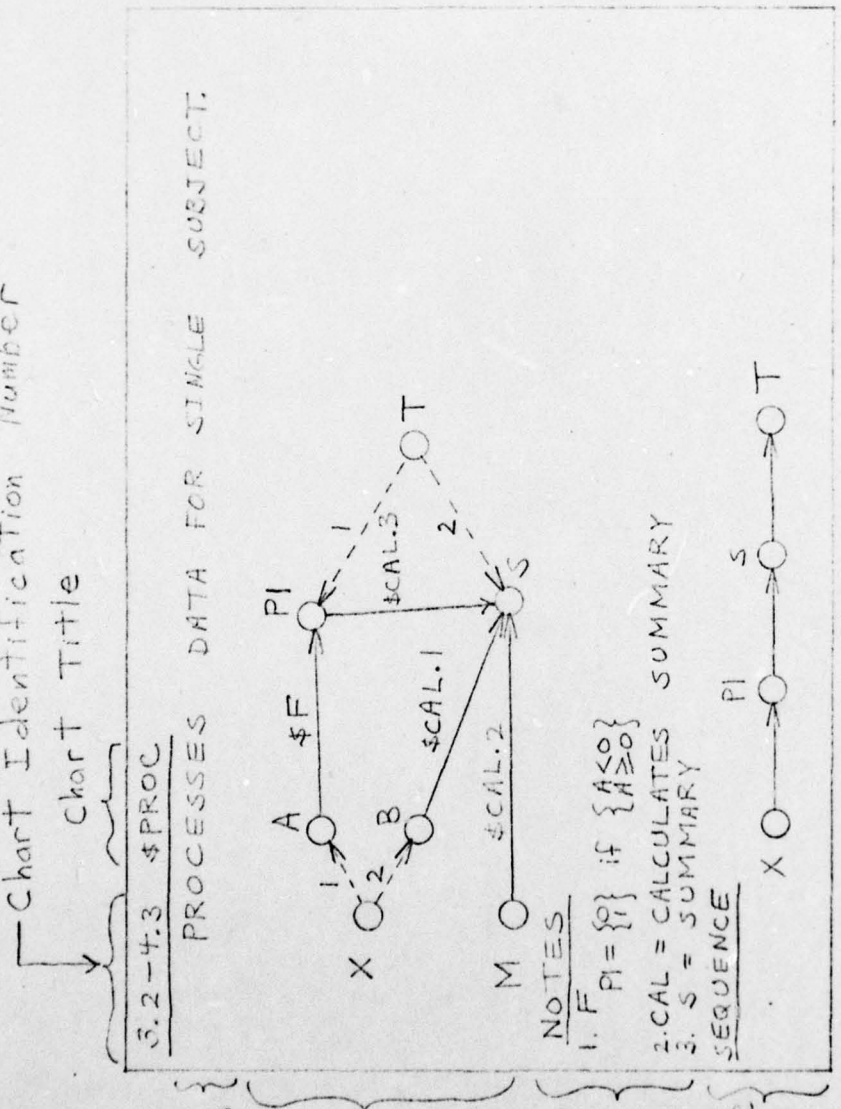


Figure 11. Typical Functional Development Chart

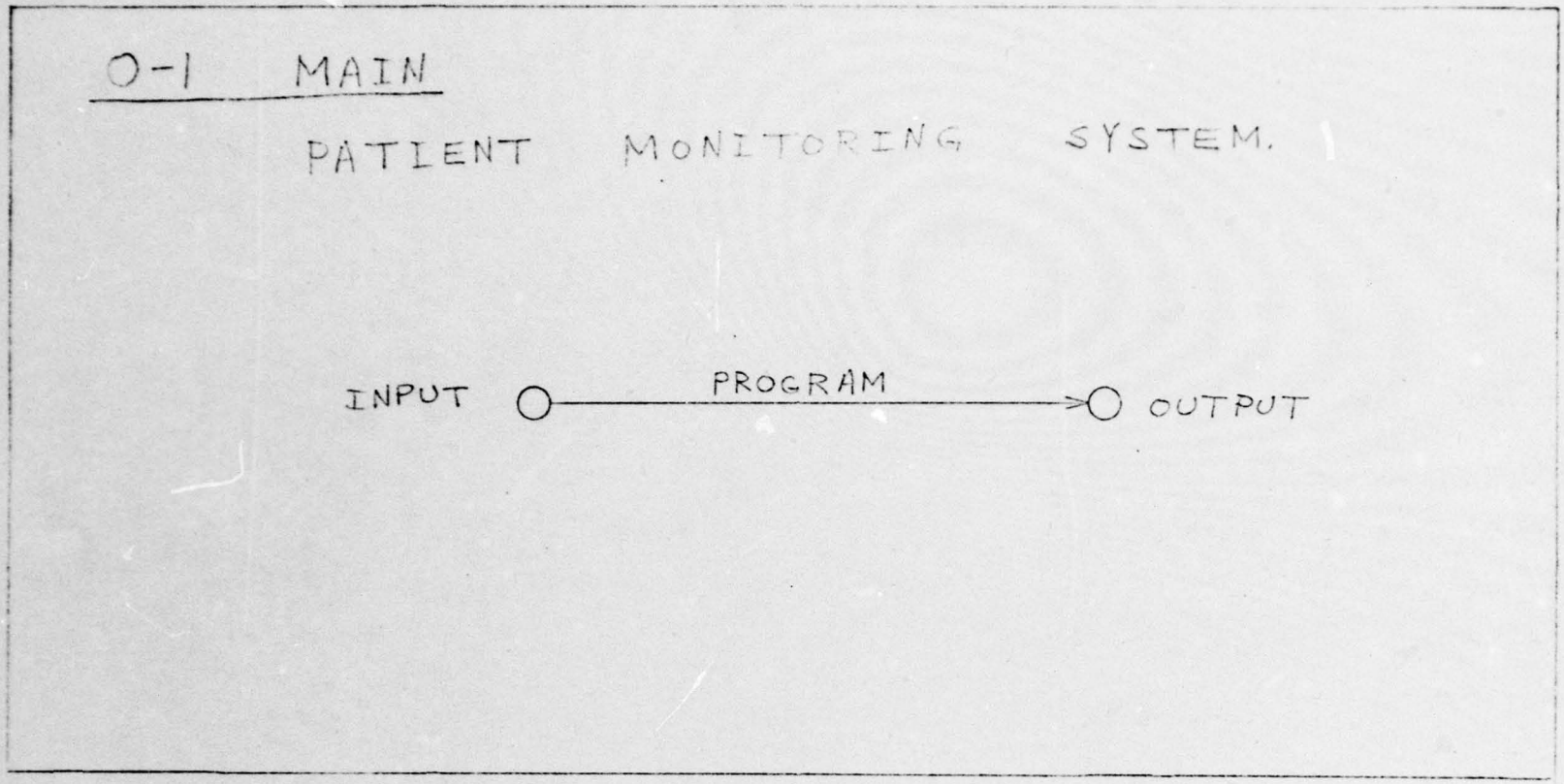
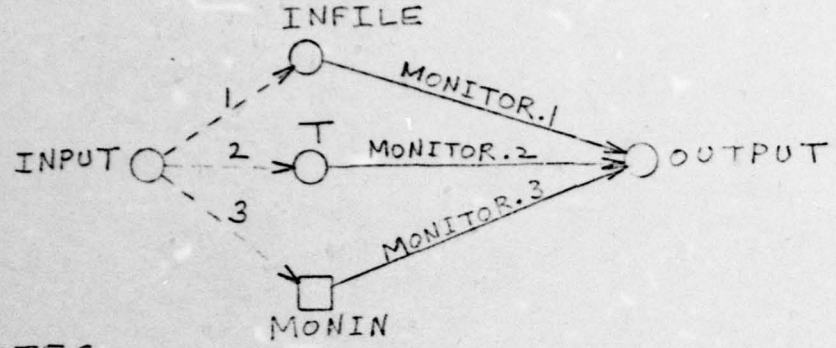


Figure 12 First (Top)-level FDC.

1-2 PROGRAM

MONITORS INTENSIVE-CARE PATIENTS.



NOTES

1. INFILE = INTERNAL FILE CONTAINS MONITORING INTERVAL AND TIME OF LAST MEASUREMENT OF PATIENT'S FACTORS FOR EACH PATIENT IN INTENSIVE CARE UNIT. ALSO CONTAINS OFF-ON INDICATOR FOR EACH BED. MONITORING INTERVAL AND OFF-ON STATUS ARE SET BY SEPARATE PROGRAM WHICH INTERRUPTS THIS PROGRAM WHEN VALUES ARE CHANGED ON MONITOR UNIT.
2. T = TIME FROM REAL-TIME CLOCK.
3. MONIN = MONITOR INPUTS.

Figure 13. Second-level FDC.

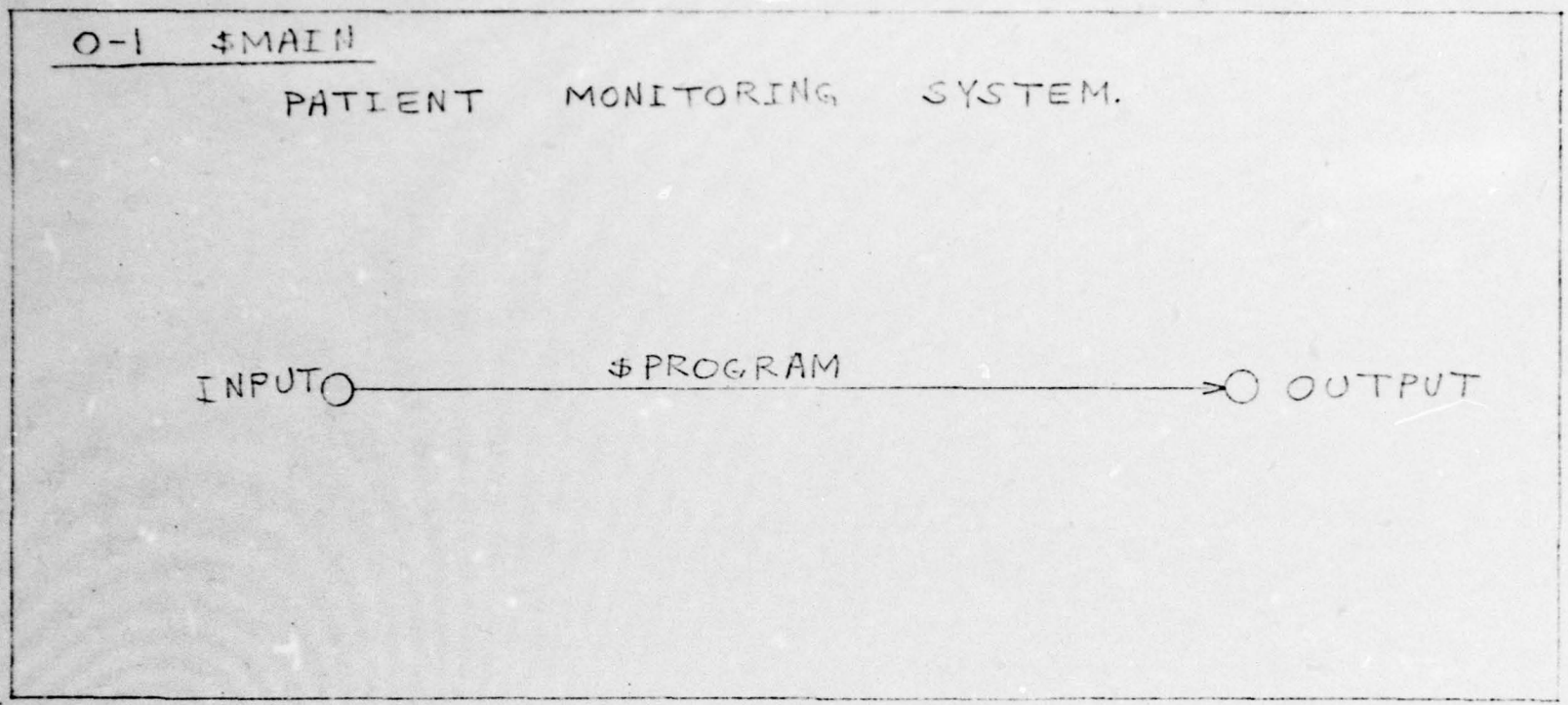


Figure 14 Final form of top-level FDC.

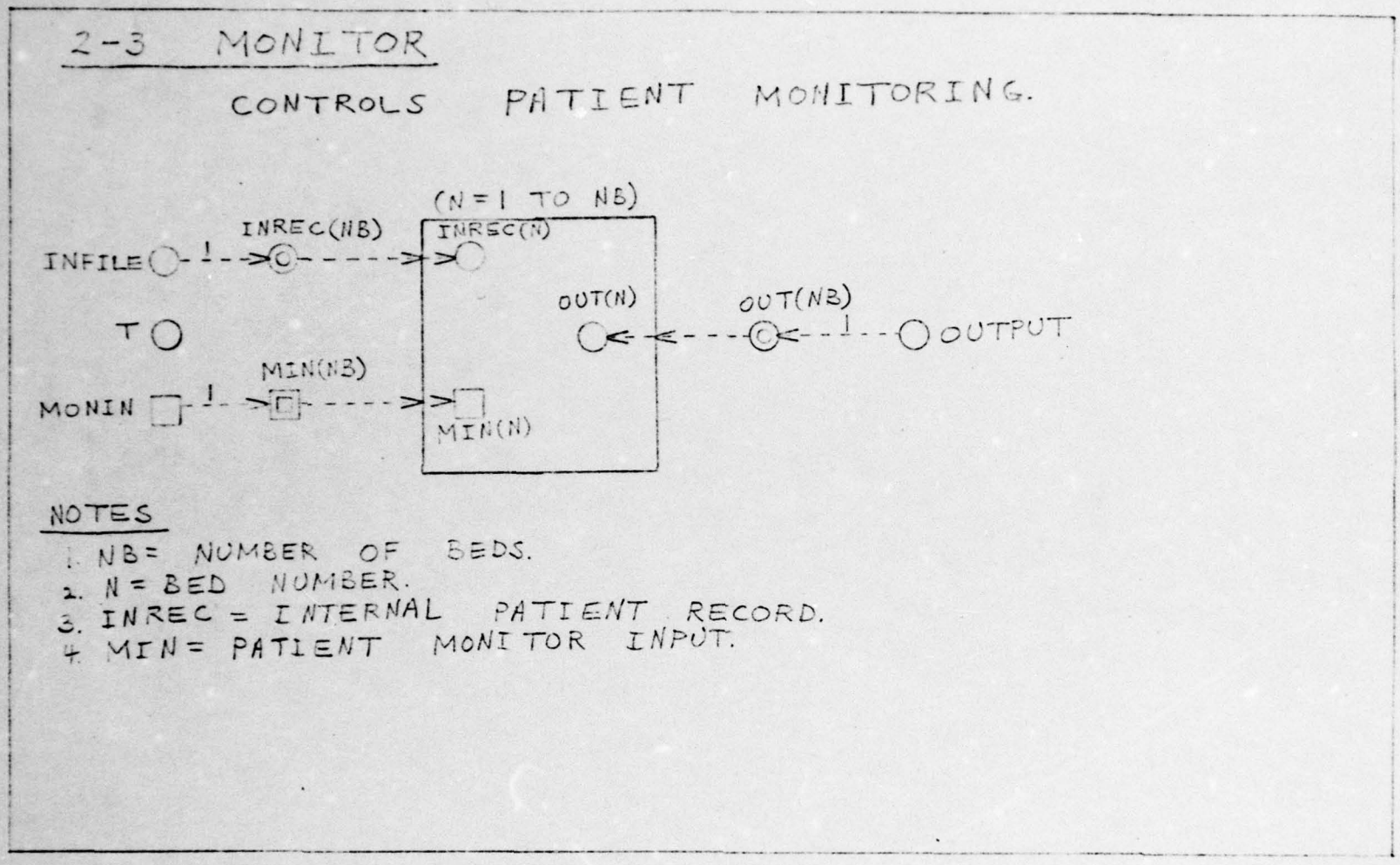


Figure 15 Partial third-level FDC at end of first (data development) step.

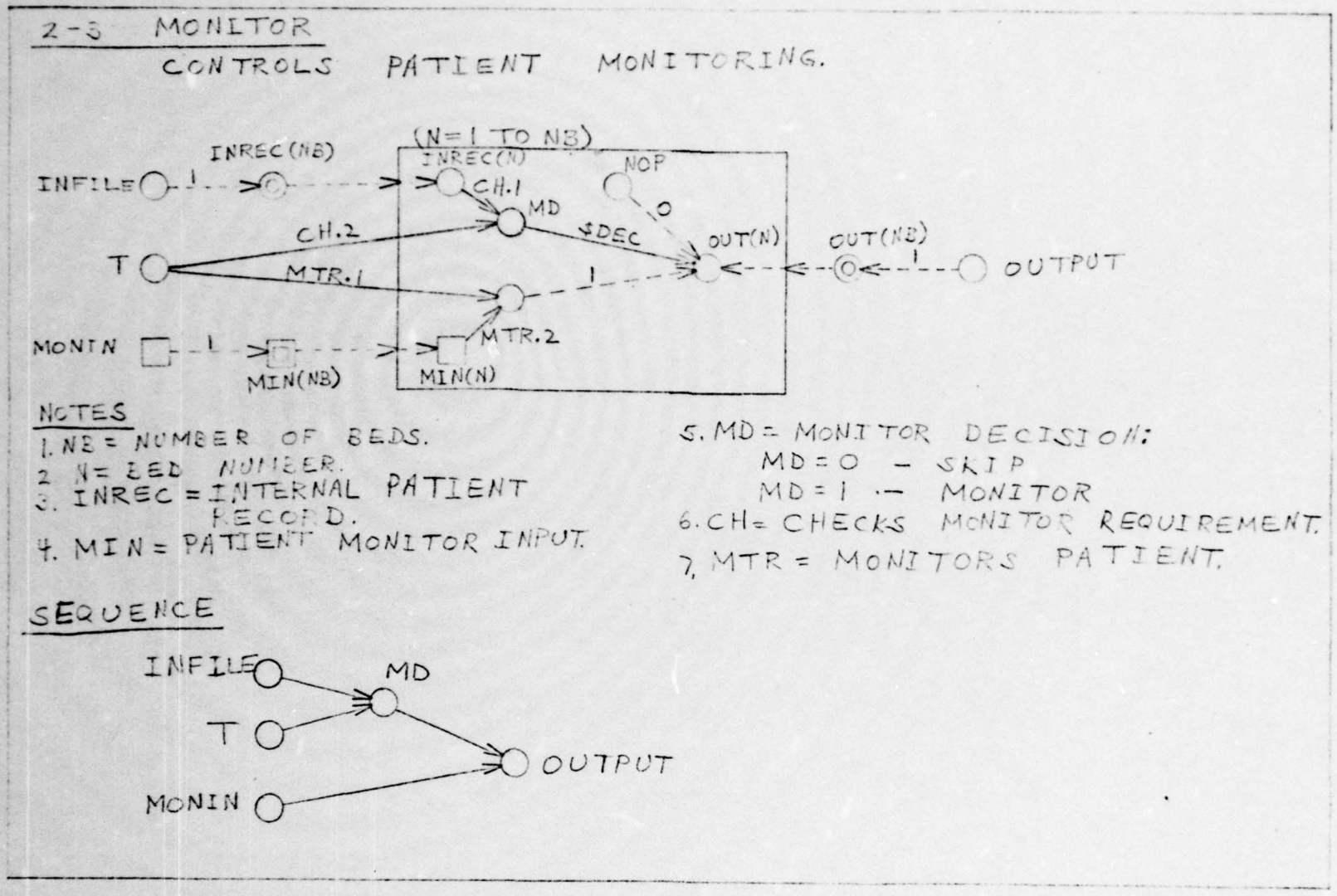
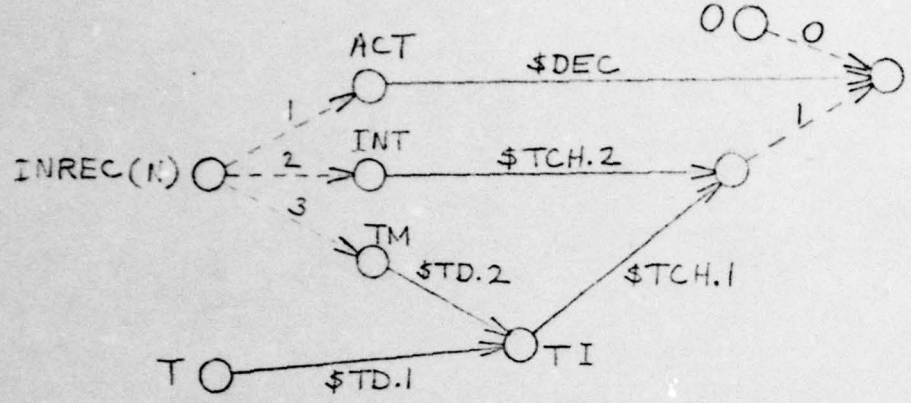


Figure 16. Complete third-level FDC.

3-4.1 $CH

CHECKS MONITOR PERIOD.



NOTES

1. TD = TIME DURATION:
   TI = T − TM; IF TI ≤ 0 THEN TI = TI + 86400.
2. TCH = TIME CHECK:
   IF TI ≥ INT THEN MD = 1; ELSE MD = 0;

SEQUENCE

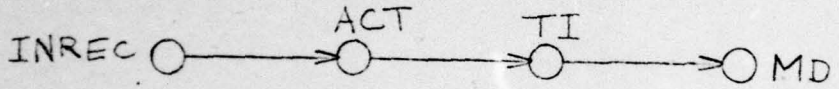


Figure 17. FDC for CH

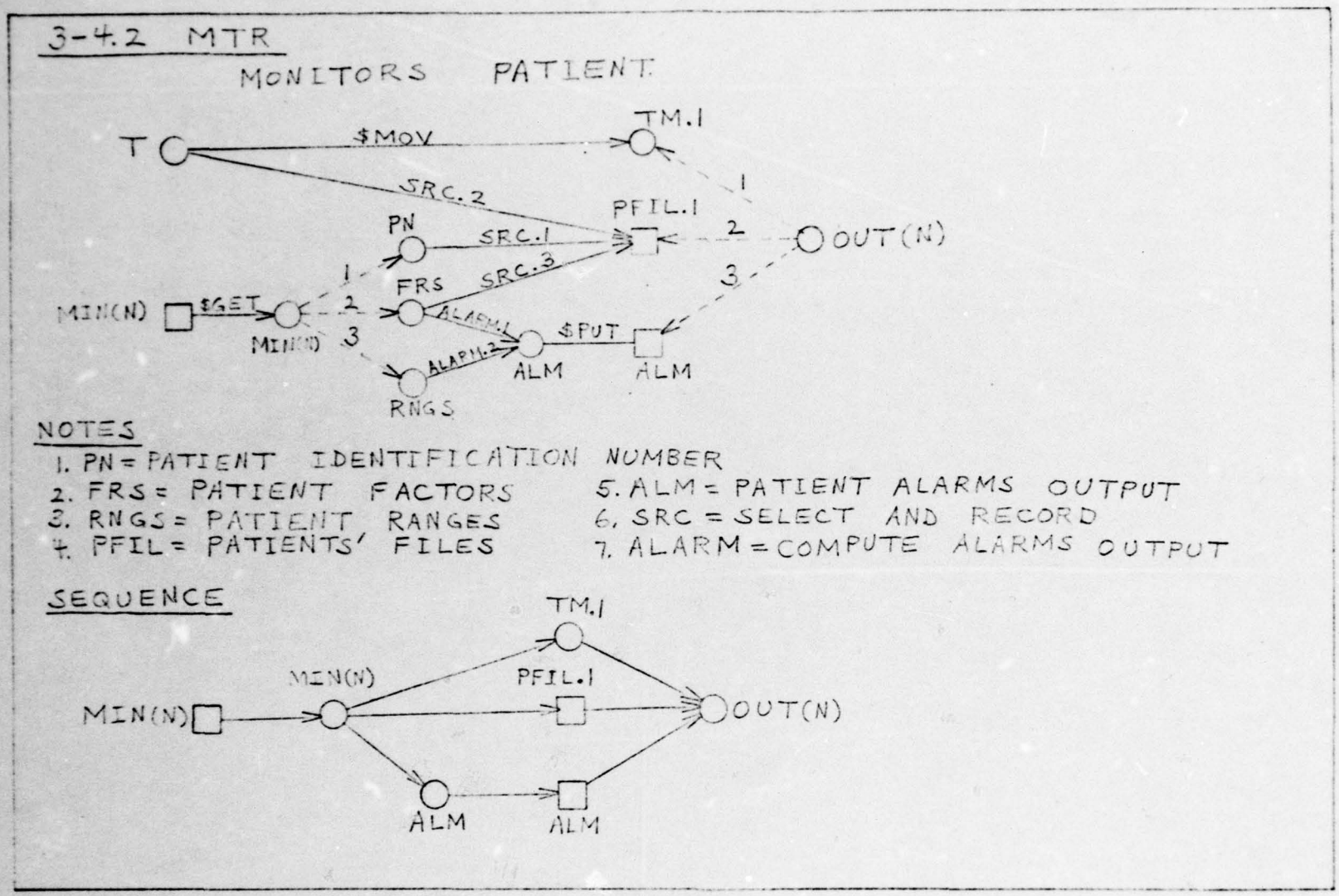
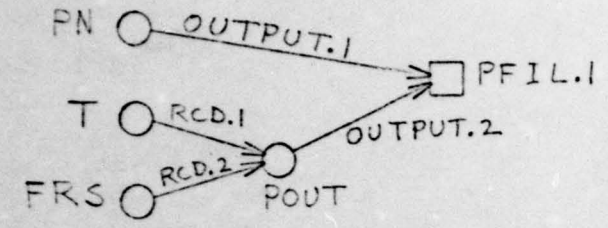


Figure 18. FDC for MTR.

4.2 - 5.1 SRC

SELECTS PATIENT SUBFILE AND RECORDS FACTORS.



NOTES

1. PREC = PATIENT RECORDS.
2. RCD = RECORD.
3. OUTPUT IS A BASIS LANGUAGE SUBPROGRAM; NOT A F-FUNCTION. IT OUTPUTS POUT TO THE SUBFILE FOR PATIENT PN.

SEQUENCE

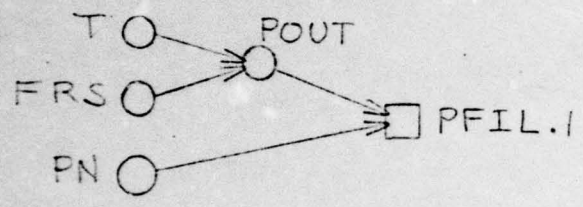


Figure 19. FDC for SRC

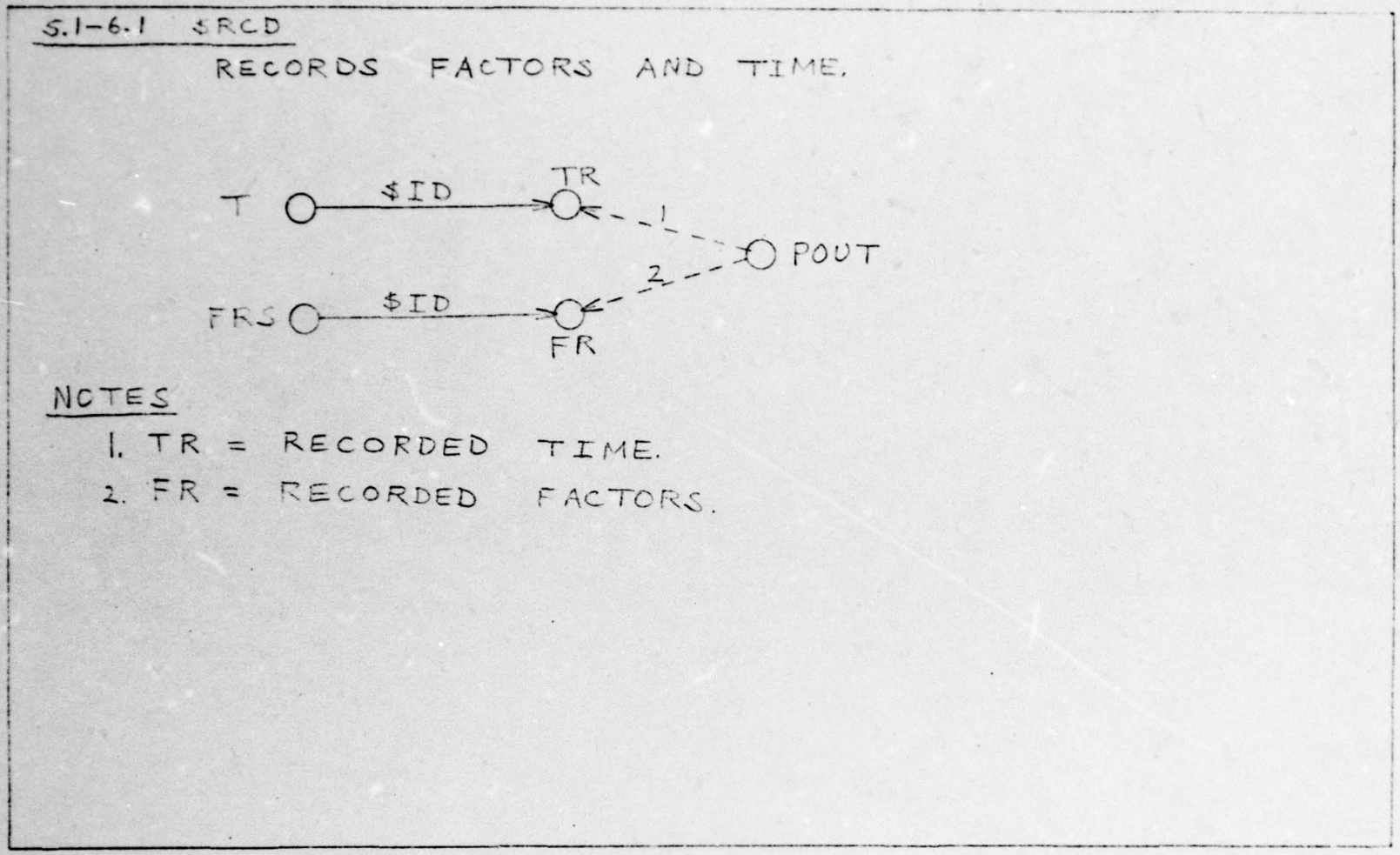
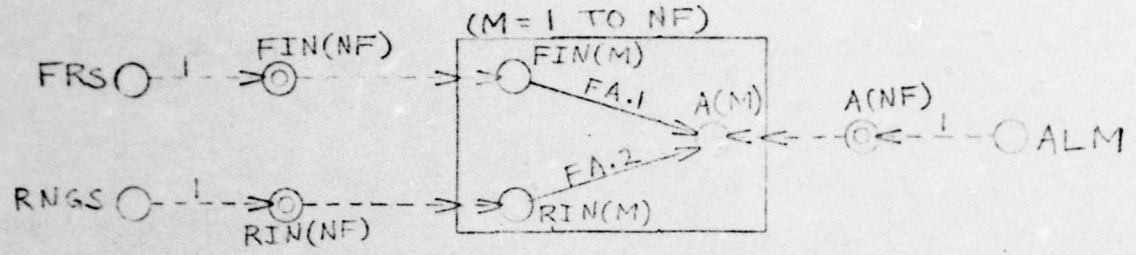


Figure 20. FDC for RCD.

4.2-5.2 ALARM

COMPUTES PATIENT ALARM OUTPUT.



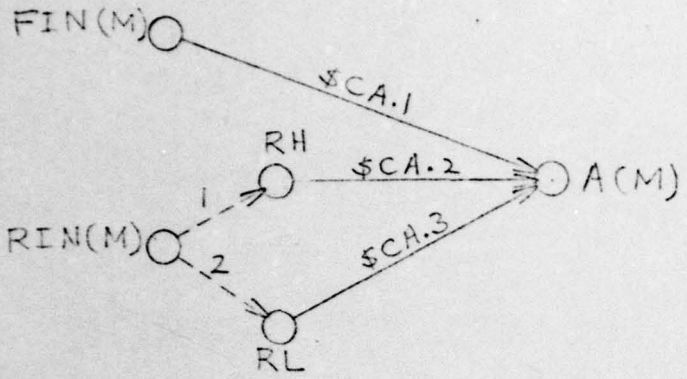
NOTES

1. FIN = INPUT FACTOR.
2. RIN = INPUT RANGE.
3. NF = NUMBER OF FACTORS.
4. A = FACTOR ALARM.
5. M = INDEX.
6. FA = COMPUTE FACTOR ALARM.

Figure 21. FDC for ALARM.

5.2-6.2 $FA

COMPUTES FACTOR ALARM.



NOTES

1. RH = HIGH RANGE.
2. RL = LOW RANGE.
3. CA = COMPUTE ALARM:
   IF (RL ≤ FIN(M) ∧ FIN(M) ≤ RH) THEN A(M) = 0;
   ELSE A(M) = 1;

Figure 22. FDC for FA.

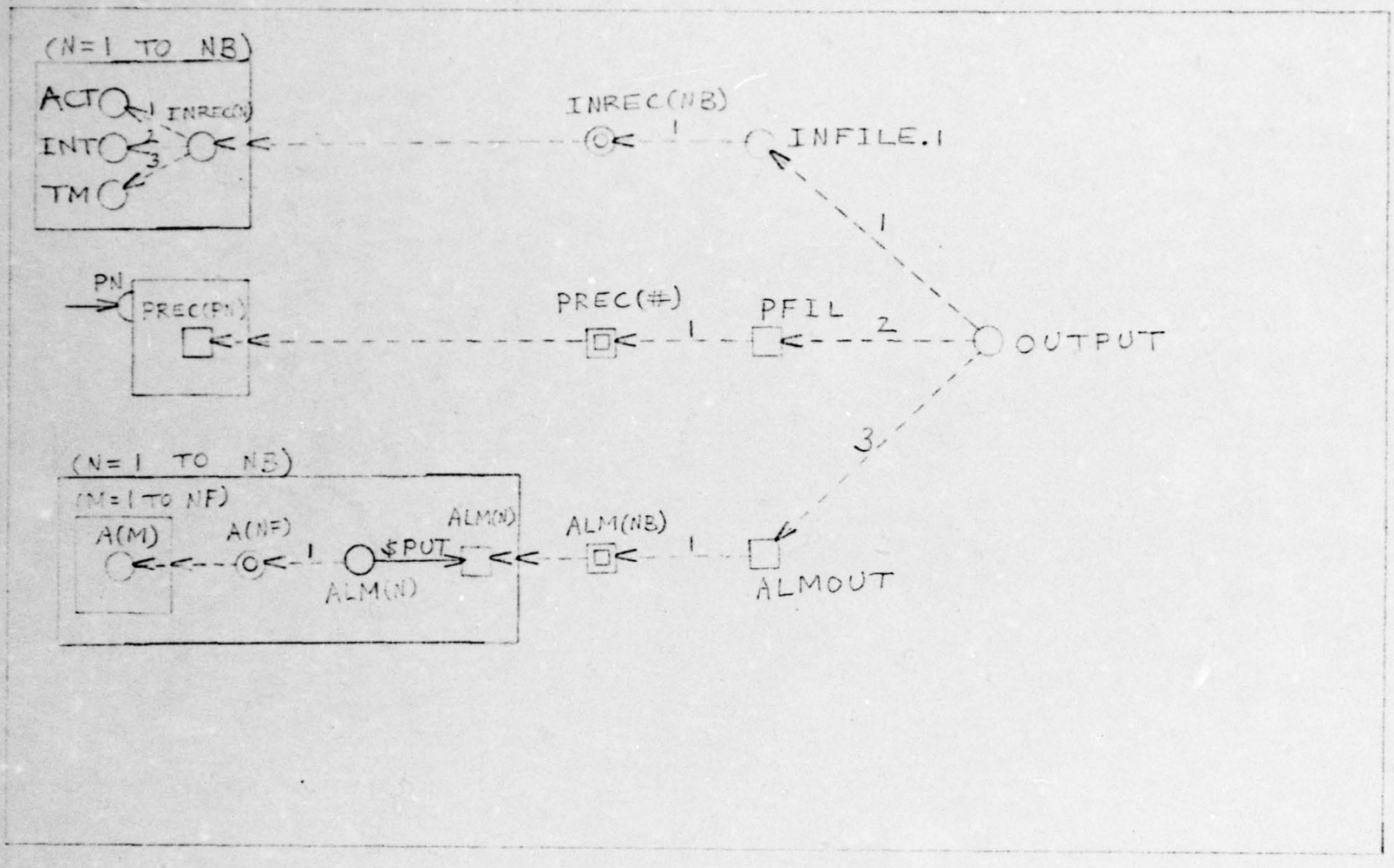
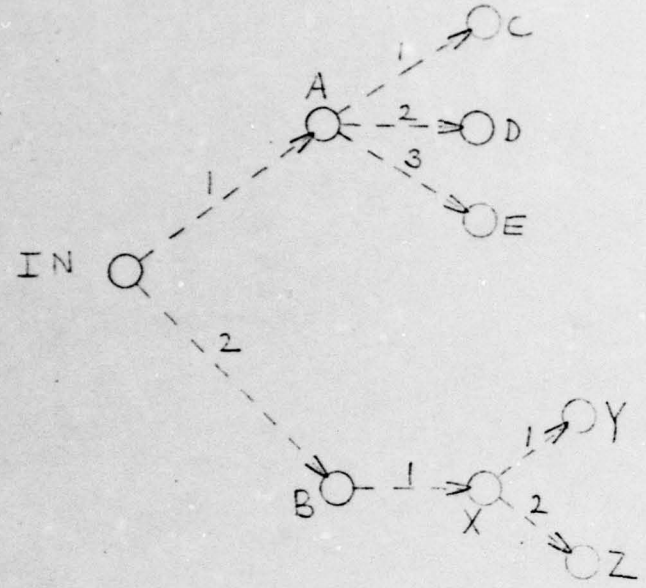


Figure 23. Physical data structure of OUTPUT.



(a)

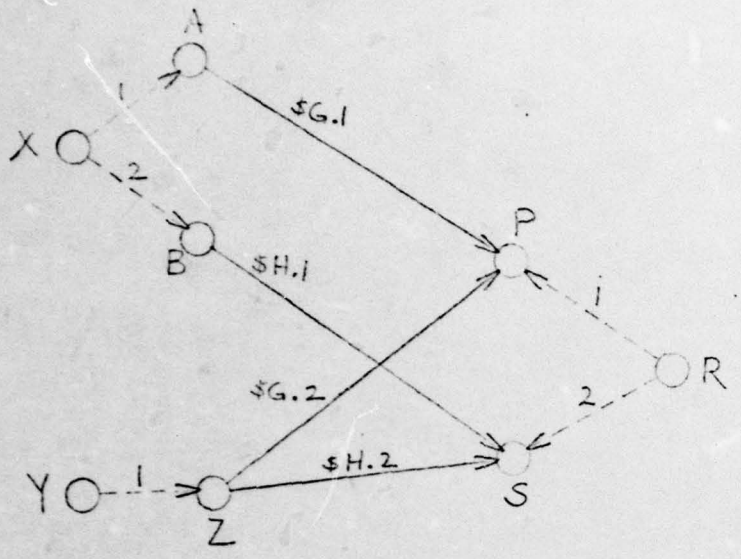
```
DAD    1.1  IN,
         2.1  A,
           3.1  C,
           3.2  D,
           3.3  E,
         2.2  B,
           3.1  X,
             4.1  Y,
             4.2  Z;
```

(b)

Figure 24. Data structures.

(a) FDC-type representation;
(b) DAD statement.

Figure 25. Argument notation.
(a) FDC segment; (b) argument values for function F.



$0 = R
$1 = X
$2 = Y
} from Fig. 25b

$0.1 = P
$0.2 = S
$1.1 = A
$1.2 = B
$2.1 = Z

(b)

Figure 26. Argument component notation.

(a) FDC segment; (b) argument component values for function F invoked in Fig. 24a.